[MS-IMSA]:
Internet Information Services (IIS) IMSAdminBaseW Remote Protocol Specification

Revision Summary

Date	Revision History	Revision Class	Comments
07/20/2007	0.1	Major	MCPP Milestone 5 Initial Availability
09/28/2007	0.1.1	Editorial	Revised and edited the technical content.
10/23/2007	0.1.2	Editorial	Revised and edited the technical content.
11/30/2007	0.2	Minor	Updated the technical content.

Date	Revision History	Revision Class	Comments
01/25/2008	0.2.1	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

The Internet Information Services (IIS) IMSAdminBaseW Remote Protocol defines interfaces that provide Unicode-compliant methods for remotely accessing and administering the IIS **metabase** associated with an application that manages IIS configuration, such as the IIS snap-in for **Microsoft Management Console (MMC)**.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

ASCII
Decrypting
Dynamic Endpoint
Encryption
Endpoint
Globally Unique Identifier (GUID)
HRESULT
Interface Definition Language (IDL)
Man in the Middle (MITM)
MD5 Hash
Microsoft Management Console (MMC)
Network Data Representation (NDR)
Opnum
Plaintext
Private Key
Public Key
Remote Procedure Call (RPC)
RPC Protocol Sequence
RPC Transport
Session Key
Unicode
Universally Unique Identifier (UUID)
Well-Known Endpoint

The following terms are specific to this document:

Cipher Text: A message that has been encrypted.

Cleartext: In cryptography, cleartext is the form of a message (or data) that is transferred or stored without cryptographic protection.

Decryption: The process of converting **cipher text** to **plaintext**. Decryption is the opposite of **encryption**.

Key Exchange Key Pair: A public/private key pair used to encrypt **session keys** so that they can be safely stored and exchanged with other users. For more information, see [PUBKEY].

Key Exchange Public Key: The **public key** of a **key exchange key pair**.

Key Exchange Private Key: The **private key** of the **key exchange key pair**.

Metabase: The name of the configuration storage implemented by Microsoft Internet Information Services (IIS).

RC4: A data **encryption** algorithm based on the RC4 symmetric stream cipher, as specified in [RC4-CRYPTO].

RSA: RSA Data Security, Inc., a major developer and publisher of **public key** cryptography standards (PKCS).

RSA Public Key algorithm: A key exchange and signature algorithm based on the popular **RSA Public Key** cipher.

Secure Session: An active communication channel that has associated cryptographic keys and possibly other state.

Signature Key Pair: The public/private key pair used for authenticating (digitally signing) messages. For more information, see [PUBKEY].

Signature Private Key: The **private key** of a **signature key pair**.

Signature Public Key: The **public key** of a **signature key pair**.

Signed Hash: A hash signed with a **signature private key**.

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

1.2 References

1.2.1 Normative References

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

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

[MS-DCOM] Microsoft Corporation, "Distributed Component Object Model (DCOM) Remote Protocol Specification", March 2007.

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

[MS-ERREF] Microsoft Corporation, "Windows Error Codes", January 2007.

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

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions", January 2007.

[RC4-CRYPTO] RSA Laboratories, "Crypto FAQ: Chapter 3 Techniques in Cryptography: 3.6 Other Cryptographic Techniques: 3.6.3 What Is RC4?", http://www.rsa.com/rsalabs/node.asp?id=2250

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC3447] Jonsson, J. and Kaliski, B., "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003, http://www.ietf.org/rfc/rfc3447.txt

1.2.2 Informative References

[MSDN-CoInitialize] Microsoft Corporation, "CoInitialize," http://msdn2.microsoft.com/en-us/library/ms678543.aspx

[MSDN-CRYPTO] Microsoft Corporation, "Cryptography Reference", http://msdn2.microsoft.com/en-us/library/aa380256.aspx

[PUBKEY] RSA Laboratories, "Crypto FAQ: Chapter 2 Cryptography: 2.1 Cryptographic Tools: 2.1.1 What Is Public-Key Cryptography?", http://www.rsa.com/rsalabs/node.asp?id=2165

1.3 Protocol Overview (Synopsis)

The Internet Information Services (IIS) IMSAdminBaseW Remote Protocol is a client/server protocol that is used for remotely managing a hierarchical configuration data store (metabase). The layout and specifics of such a store are specified in section 3.1.1.

A remote metabase management session begins with the client initiating the connection request to the server. If the server grants the request, the connection is established. The client can then make multiple requests to read or modify the metabase on the server by using the same session until the session is terminated.

A typical remote metabase management session involves the client connecting to the server and requesting to open a metabase node on the server. If the server accepts the request, it responds with an **RPC** context handle that refers to the node. The client uses this RPC context handle to operate on that node. This usually involves sending another request to the server specifying the type of operation to perform and any specific parameters that are associated with that operation. If the server accepts this request, it attempts to change the state of the node based on the request and responds to the client with the result of the operation. When the client is finished operating on the server nodes, it terminates the protocol by sending a request to close the RPC context handle.

1.4 Relationship to Other Protocols

The IIS IMSAdminBaseW Remote Protocol relies on the Distributed Component Object Model (DCOM) Remote Protocol, which uses RPC as a **transport**, as specified in [MS-DCOM].

No other IIS protocols rely on this protocol.

1.5 Prerequisites/Preconditions

This protocol is implemented over DCOM and RPC and, as a result, has the prerequisites identified in [MS-DCOM] and [MS-RPCE] as being common to DCOM and RPC interfaces.

The IIS IMSAdminBaseW Remote Protocol assumes that a client has obtained the name of a server that supports this protocol suite before the protocol is invoked.

1.6 Applicability Statement

This protocol is applicable when an application needs to remotely configure an IIS server.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

Supported Transports: The IIS IMSAdminBaseW Remote Protocol uses the DCOM Remote Protocol and multiple **RPC protocol sequences**, as specified in section 2.1.

Protocol Versions: This protocol has multiple interfaces, as specified in section 3.

Security and Authentication Methods: Authentication and security are provided as specified in [MS-DCOM] and [MS-RPCE].

Capability Negotiation: The IIS IMSAdminBaseW Remote Protocol does not support negotiation of the interface version to use. Instead, this protocol uses only the interface version number specified in the **IDL** for versioning and capability negotiation.

1.8 Vendor-Extensible Fields

The IIS IMSAdminBaseW Remote Protocol does not have any vendor-extensible fields.

1.9 Standards Assignments

The following parameters are private Microsoft assignments.

Parameter	Value	Reference
RPC Interface **UUID** for **IMSAdminBaseW**	70B51430-B6CA-11d0-B9B9-00A0C922E750	None
RPC Interface UUID for **IMSAdminBase2W**	8298d101-f992-43b7-8eca-5052d885b995	None
RPC Interface UUID for **IMSAdminBase3W**	f612954d-3b0b-4c56-9563-227b7be624b4	None

2 Messages

The following sections specify how IIS IMSAdminBaseW Remote Protocol messages are transported and common data types.

2.1 Transport

The Transport protocol MUST use the DCOM Remote Protocol, as specified in [MS-DCOM], as its transport. On its behalf, the DCOM Remote Protocol uses the following RPC protocol sequence: RPC over TCP, as specified in [MS-RPCE]. This protocol uses RPC **Dynamic Endpoints**, as specified in [C706] section 4.

This protocol MUST use the following UUIDs:

IMSAdminBaseW: 70B51430-B6CA-11D0-B9B9-00A0C922E750

IMSAdminBase2W: 8298D101-F992-43B7-8ECA-5052D885B995

IMSAdminBase3W: F612954D-3B0B-4C56-9563-227B7BE624B4

2.2 Common Data Types

In addition to RPC base types and definitions specified in [C706] and [MS-DTYP], additional data types are defined below.

All multibyte integer values in the messages declared in this section are stored using the little-endian byte order.

2.2.1 IIS_CRYPTO_BLOB

The **IIS_CRYPTO_BLOB** message defines a block of data, possibly encrypted, that is transferred between client and server. It is used to transfer **public keys**, hash information, encrypted and cleartext data.

```
typedef struct {
  DWORD BlobSignature;
  DWORD BlobDataLength;
  [size_is(BlobDataLength)] unsigned char BlobData[*];
} IIS_CRYPTO_BLOB;
```

BlobSignature: The structure signature for this binary large object (BLOB).

Value	Meaning
SESSION_KEY_BLOB_SIGNATURE 0x624b6349	The **BlobData** contains the **session key** used to encrypt sensitive data exchanged between client and server. See SESSION_KEY_BLOB (section 2.2.1.2) for more information about the **BlobData** layout.
PUBLIC_KEY_BLOB_SIGNATURE 0x62506349	The **BlobData** contains the public key for a particular IIS **encryption** behavior. See PUBLIC_KEY_BLOB (section 2.2.1.1) for more information about the **BlobData** layout.

Value	Meaning
ENCRYPTED_DATA_BLOB_SIGNATURE 0x62446349	The **BlobData** contains encrypted data. See ENCRYPTED_DATA_BLOB (section 2.2.1.5) for more information about the **BlobData** layout.
HASH_BLOB_SIGNATURE 0x62486349	The **BlobData** contains a hash. See HASH_BLOB (section 2.2.1.3) for more information about the **BlobData** layout.
CLEARTEXT_DATA_BLOB_SIGNATURE 0x62436349	The **BlobData** contains clear text data. See CLEARTEXT_DATA_BLOB (section 2.2.1.4) for more information about the **BlobData** layout.

BlobDataLength: The size, in bytes, of **BlobData**.

BlobData: A block of bytes that can be interpreted based on **BlobSignature**.

2.2.1.1 PUBLIC_KEY_BLOB

The PUBLIC_KEY_BLOB message is used to store information about **RSA Key Exchange Public Keys** and RSA **Signature Public Keys**. It is used during **secure session** negotiation.

The syntax of the PUBLIC_KEY_BLOB message is represented by the following diagram.

Type (1 byte): An 8-bit unsigned integer. This field MUST be set to 0x6. This indicates that the public key is transferred.

Version (1 byte): An 8-bit unsigned integer. This field MUST be set to 0x2.

Reserved (2 bytes): A 16-bit unsigned integer. This field MUST be set to 0x0.

AlgID (4 bytes): A 32-bit unsigned integer. This field is set to the CALG_RSA_KEYX value if the key exchange public key is stored in the BLOB or the CALG_RSA_SIGN value if the signature public key is stored.

Value	Meaning
CALG_RSA_KEYX 0x0000A400	RSA public key exchange algorithm
CALG_RSA_SIGN 0x00002400	RSA public key signature algorithm

Magic (4 bytes): A 32-bit unsigned integer. This field MUST be set to 0x31415352. The value can be interpreted as the ASCII-encoded string "RSA1".

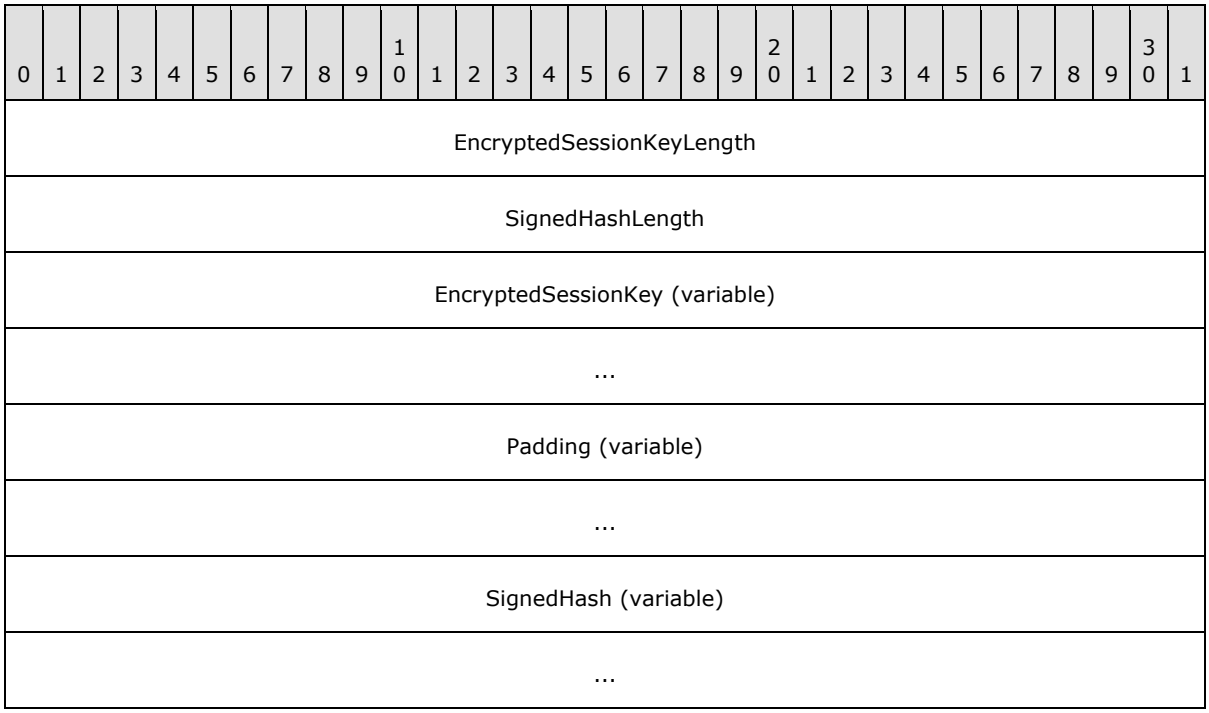
BitLen (4 bytes): A 32-bit unsigned integer that specifies the size of the public key in bits. This field MUST be set to 0x200 (512) because the 512 (=0x200) bit RSA key is used.

PubExp (4 bytes): A 32-bit unsigned integer that is a public exponent, as specified in [RFC3447].

Modulus (variable): A variable-length array of bytes that stores the RSA public key. The size, in bytes, of the **Modulus** field is **BitLen**/8.

2.2.1.2 SESSION_KEY_BLOB

The SESSION_KEY_BLOB is used to store session keys that are transferred during the secure session negotiation.



EncryptedSessionKeyLength (4 bytes): A 32-bit unsigned integer that contains the size, in bytes, of the **EncryptedSessionKey** field.

SignedHashLength (4 bytes): A 32-bit unsigned integer that contains the size, in bytes, of the **SignedHash** field.

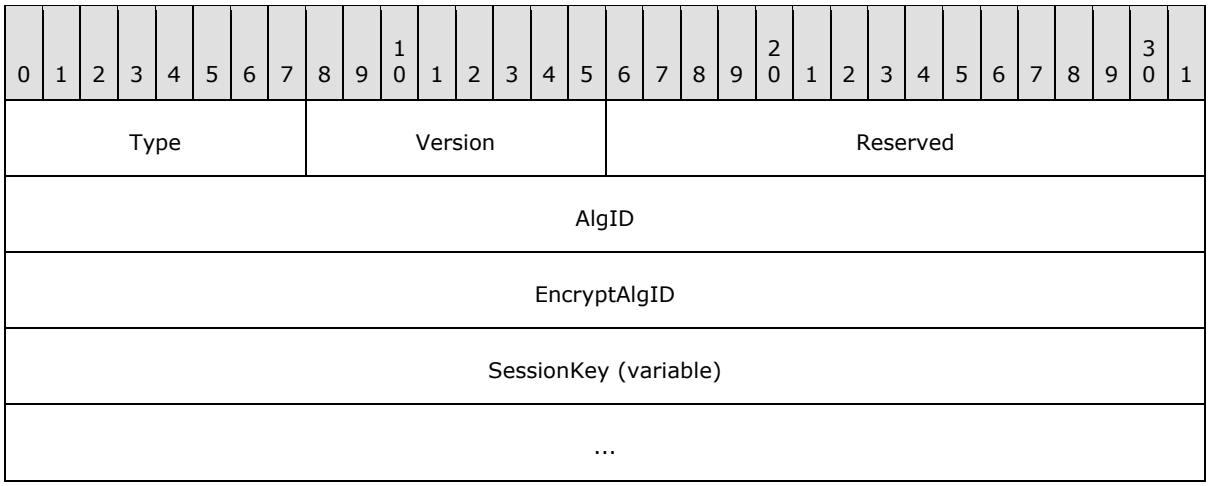
EncryptedSessionKey (variable): A variable-length array of bytes that contains session key information. For more information about the internal organization of data inside this field, see ENCRYPTED_SESSION_KEY_BLOB (section 2.2.1.2.1).

Padding (variable): A variable-length array of bytes that contains zero to eight bytes of padding based on the **SessionKeyDataLength** field. The number of padding bytes is calculated as the difference between an 8-byte aligned **EncryptedSessionKeyLength** field and the actual **EncryptedSessionKeyLength** field.

SignedHash (variable): A variable-length array of bytes that contain the **signed hash** of the session key.

2.2.1.2.1 ENCRYPTED_SESSION_KEY_ BLOB

The ENCRYPTED_SESSION_KEY_BLOB message layout is described in the following diagram.



Type (1 byte): An 8-bit unsigned integer that specifies that the session key is transferred. This field MUST be set to 0x6.

Version (1 byte): An 8-bit unsigned integer value. This field MUST be set to 0x2.

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

AlgID (4 bytes): A 32-bit unsigned integer. This field MUST be set to the CALG_RC4 value, which MUST be used to indicate that the RC4 stream encryption algorithm will be used for the data encryption, as specified in [RC4-CRYPTO].

Value	Meaning
CALG_RC4 0x0000a400	The RC4 stream encryption algorithm.

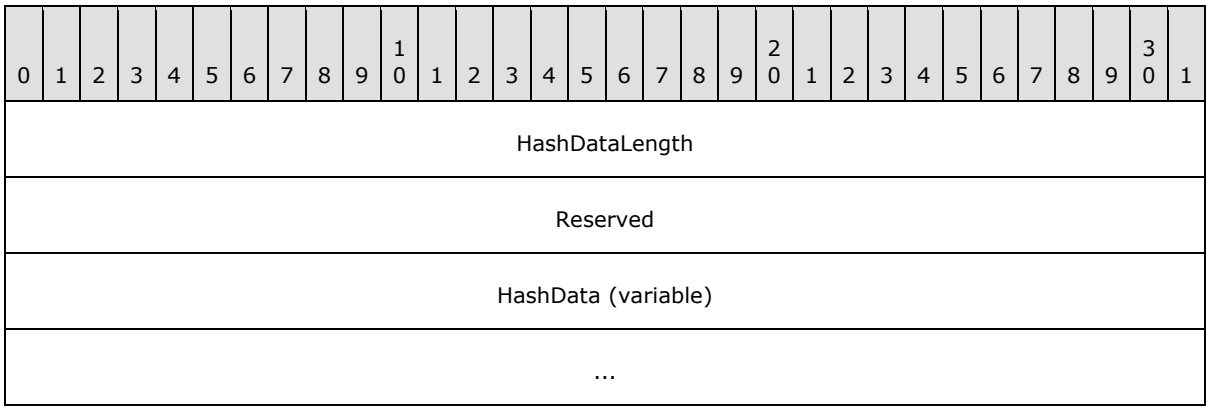
EncryptAlgID (4 bytes): An unsigned 32-bit integer that MUST be set to the CALG_RSA_KEYX value, which indicates that the session key was encrypted using the **RSA public key algorithm**.

Value	Meaning
CALG_RSA_KEYX 0x0000a400	The RSA public key algorithm.

SessionKey (variable): A variable-length array of bytes that contains the actual session key of **AlgID** type, which is encrypted by the algorithm specified by **EncryptAlgID**. The size of the **SessionKey** field is always the same as the size of the modulus of the public key used for encryption.

2.2.1.3 HASH_BLOB

The HASH_BLOB message stores the hash that is exchanged during the secure session negotiation.



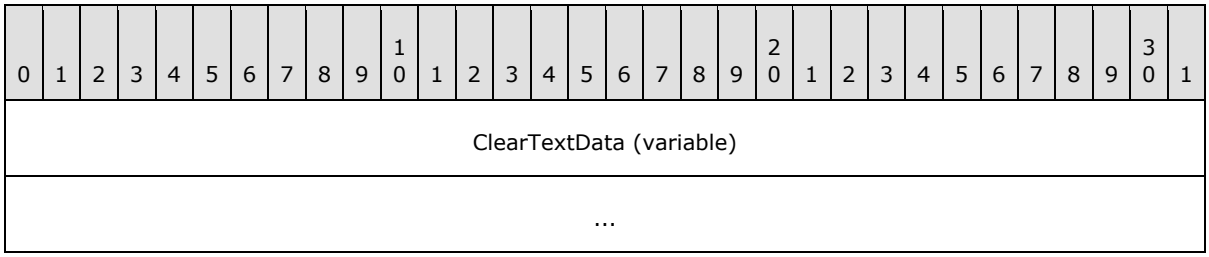
HashDataLength (4 bytes): A 32-bit unsigned integer that stores the size, in bytes, of the **HashData** field.

Reserved (4 bytes): This field MUST be set to 0x00000000 and MUST be ignored on receipt.

HashData (variable): A variable-length array that contains the hash.

2.2.1.4 CLEARTEXT_DATA_BLOB

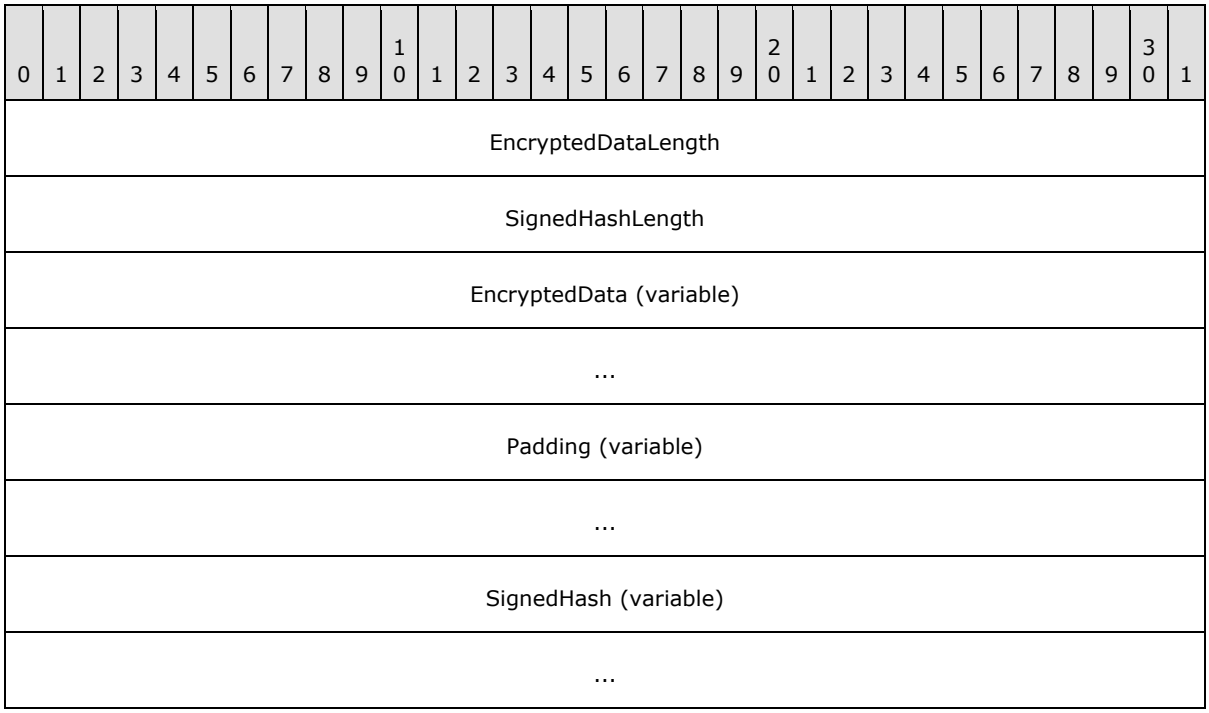
The CLEARTEXT_DATA_BLOB message stores **cleartext** data that does not need encryption, but uses the **IIS_CRYPTO_BLOB** message to store the data.



ClearTextData (variable): A variable-length array of bytes that contains cleartext data.

2.2.1.5 ENCRYPTED_DATA_BLOB

The ENCRYPTED_DATA_BLOB message stores the encrypted sensitive data that is transferred between client and server.



EncryptedDataLength (4 bytes): A 32-bit unsigned integer that stores the size, in bytes, of the **EncryptedData** field.

SignedHashLength (4 bytes): A 32-bit unsigned integer that stores the size, in bytes, of the **SignedHash** field.

EncryptedData (variable): A variable-length array of bytes containing encrypted data. The cleartext data before the encryption is stored in CLEARTEXT_WITH_PREFIX_BLOB format.

Padding (variable): A variable-length array of bytes where the length of the padding is based on the **EncryptedDataLength** field. The number of padding bytes is calculated as the difference between the 8-byte aligned **EncryptedDataLength** field and the actual **EncryptedDataLength** field.

SignedHash (variable): A variable-length array of bytes that contains the signed hash of the **EncryptedData** field.

2.2.1.5.1 CLEARTEXT_WITH_PREFIX_BLOB

The CLEARTEXT_WITH_PREFIX_BLOB is used to store cleartext data before it is encrypted and serialized into the **BlobData** field of the **IIS_CRYPTO_BLOB** message with the **BlobSignature** field set to ENCRYPTED_DATA_BLOB_SIGNATURE.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Reserved																															
ClearTextData (variable)																															
...																															

Reserved (4 bytes): This field MUST be set to zero and MUST be ignored on receipt.

ClearTextData (variable): A variable-length array of bytes that contains cleartext data.

2.2.2 Secure Session Negotiation Constants

Constant/value	Description
HASH_TEXT_STRING_1 IIS Key Exchange Phase 3	The constant string used to calculate the hash sent by the client with the **R_KeyExchangePhase2** call.
HASH_TEXT_STRING_2 IIS Key Exchange Phase 4	The constant string used to calculate the hash sent by the server in response to the **R_KeyExchangePhase2** call.

2.2.3 METADATA_GETALL_RECORD

The **METADATA_GETALL_RECORD** structure defines an analogous structure to **METADATA_RECORD**, but is used only to return data from a call to the **R_GetAllData** method. Data retrieval specifications are provided in **R_GetAllData** method parameters, not in this structure (as is the case with **METADATA_RECORD**). The **R_GetAllData** method returns the data from multiple entries as an array of **METADATA_GETALL_RECORD** structures.

```
typedef struct {
  DWORD dwMDIdentifier;
  DWORD dwMDAttributes;
  DWORD dwMDUserType;
  DWORD dwMDDataType;
  DWORD dwMDDataLen;
  union {
    DWORD dwMDDataOffset;
    unsigned char* pbMDData;
  };
  DWORD dwMDDataTag;
} METADATA_GETALL_RECORD,
 *PMETADATA_GETALL_RECORD;
```

dwMDIdentifier: An unsigned integer value that uniquely identifies the metabase entry.

dwMDAttributes: An unsigned integer value containing bit flags that specify how to set or get data from the metabase. This member MUST be set to a valid combination of the following values.

Value	Meaning
METADATA_INHERIT 0x00000001	In Get methods: return the inheritable data. In Set methods: the data can be inherited.
METADATA_INSERT_PATH 0x00000040	For a string data item. In Get methods: replace all occurrences of "<%INSERT_PATH%>" with the path of the data item relative to the handle. In Set methods: indicate that the string contains the Unicode character substring "<%INSERT_PATH%>".
METADATA_ISINHERITED 0x00000020	In Get methods: mark the data items that were inherited. In Set methods: not valid.
METADATA_NO_ATTRIBUTES 0x00000000	In Get methods: not applicable. Data is returned regardless of this flag setting. In Set methods: the data does not have any attributes.
METADATA_PARTIAL_PATH 0x00000002	In Get methods: return any inherited data even if the entire path is not present. This flag is valid only if METADATA_INHERIT is also set. In Set methods: not valid.
METADATA_SECURE 0x00000004	In Get methods: not valid. In Set methods: the server and client SHOULD transport and store the data in a secure fashion, as specified in 3.1.4.1.1.
METADATA_VOLATILE 0x00000010	In Get methods: not valid. In Set methods: do not save the data in long-term storage.

dwMDUserType: An unsigned integer value that specifies the user type of the data. The **dwMDUserType** member MUST be set to one of the following values.

Value	Meaning
ASP_MD_UT_APP 0x00000065	The entry contains information specific to ASP application configuration.
IIS_MD_UT_FILE 0x00000002	The entry contains information about a file, such as access permissions or logon methods.
IIS_MD_UT_SERVER 0x00000001	The entry contains information specific to the server, such as ports in use and IP addresses.
IIS_MD_UT_WAM 0x00000064	The entry contains information specific to Web Application Management (WAM).

dwMDDataType: An integer value that identifies the type of data in the metabase entry. The **dwMDDataType** member MUST be set to one of the following values.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.
BINARY_METADATA 0x00000003	Specifies binary data in any form.
DWORD_METADATA 0x00000001	Specifies all **DWORD** (unsigned 32-bit integer) data.
EXPANDSZ_METADATA 0x00000004	Specifies all data that consists of a string that includes the NULL-terminating character, and which contains environment variables that are not expanded.
MULTISZ_METADATA 0x00000005	Specifies all data represented as an array of strings, where each string includes the NULL-terminating character, and the array itself is terminated by two NULL-terminating characters.
STRING_METADATA 0x00000002	Specifies all data consisting of an ASCII string that includes the NULL-terminating character.

dwMDDataLen: An unsigned integer value that specifies the length, in bytes, of the data. If the data is a string, this value includes the ending null character. For lists of strings, this includes an additional NULL-terminating character after the final string (double NULL-terminating characters).

For example, the length of a string list containing two strings would be:

```
(wcslen(stringA) + 1) * sizeof(WCHAR) + (wcslen(stringB) + 1)
 * sizeof(WCHAR) + 1 * sizeof(WCHAR)
```

In-process clients need to specify **dwMDDataLen** only when setting binary data in the metabase. Remote clients MUST specify **dwMDDataLen** for all data types.

dwMDDataOffset: If the data was returned by value, this member contains the byte offset of the data in the buffer specified by the *pbMDBuffer* parameter of the **R_GetAllData** method. All out-of-process executions will return data by value. The array of records, excluding the data, is returned in the first part of the buffer. The data associated with the records is returned in the buffer after the array of records, and **dwMDDataOffset** is the offset to the beginning of the data associated with each record in the array.

pbMDData: A reserved member that is currently unused.

dwMDDataTag: A reserved member that is currently unused.

2.2.4 METADATA_HANDLE

The **METADATA_HANDLE** represents a node of the configuration storage tree.

This type is declared as follows:

```
typedef unsigned LONG METADATA_HANDLE, *PMETADATA_HANDLE;
```

2.2.5 METADATA_HANDLE_INFO

The **METADATA_HANDLE_INFO** structure defines information about a handle to a metabase entry.

```
typedef struct {
  DWORD dwMDPermissions;
  DWORD dwMDSystemChangeNumber;
} METADATA_HANDLE_INFO;
```

dwMDPermissions: An unsigned integer value containing the permissions with which the handle was opened. This member MUST have a valid combination of the following flags set.

Value	Meaning
METADATA_PERMISSION_READ 0x00000001	The handle can read nodes and data.
METADATA_PERMISSION_WRITE 0x00000002	The handle can write nodes and data.

dwMDSystemChangeNumber: An unsigned integer value containing the system change number when the handle was opened. This number indicates how many times changes were made to the data since the metabase was created. This value is persisted between metabase sessions. The client MAY use this information to track the changes to the metabase.

2.2.6 METADATA_RECORD

The **METADATA_RECORD** structure defines information about a metabase entry.

```
typedef struct {
  DWORD dwMDIdentifier;
  DWORD dwMDAttributes;
  DWORD dwMDUserType;
  DWORD dwMDDataType;
  DWORD dwMDDataLen;
  [unique, size_is(dwMDDataLen)]
    unsigned char* pbMDData;
  DWORD dwMDDataTag;
} METADATA_RECORD;
```

dwMDIdentifier: An unsigned integer value that uniquely identifies the metabase entry.

dwMDAttributes: An unsigned integer value containing bit flags that specify how to get or set data from the metabase. This member MUST have a valid combination of the following flags set.

Value	Meaning
METADATA_INHERIT 0x00000001	In Get methods: returns inheritable data. In Set methods: the data can be inherited.
METADATA_INSERT_PATH 0x00000040	For a string data item. In Get methods: replaces all occurrences of "<%INSERT_PATH%>" with the path of the data item relative to the handle. In Set methods: indicate that the string contains the Unicode character substring "<%INSERT_PATH%>".
METADATA_ISINHERITED 0x00000020	In Get methods: marks data items that were inherited. In Set methods: not valid.
METADATA_NO_ATTRIBUTES 0x00000000	In Get methods: not applicable. Data is returned regardless of this flag setting. In Set methods: the data does not have any attributes.
METADATA_PARTIAL_PATH 0x00000002	In Get methods: returns any inherited data even if the entire path is not present. This flag is valid only if METADATA_INHERIT is also set. In Set methods: not valid.
METADATA_SECURE 0x00000004	In Get methods: not valid. In Set methods: stores and transports the data in a secure fashion, as specified in 3.1.4.1
METADATA_VOLATILE 0x00000010	In Get methods: not valid. In Set methods: does not save the data in long-term storage.

dwMDUserType: An integer value that specifies the user type of the data. The **dwMDUserType** member MUST be set to one of the following values.

Value	Meaning
ASP_MD_UT_APP 0x00000065	The entry contains information specific to ASP application configuration.
IIS_MD_UT_FILE 0x00000002	The entry contains information about a file, such as access permissions or logon methods.
IIS_MD_UT_SERVER 0x00000001	The entry contains information specific to the server, such as ports in use and IP addresses.
IIS_MD_UT_WAM 0x00000064	The entry contains information specific to Web application management.

dwMDDataType: An unsigned integer value that identifies the type of data in the metabase entry. The **dwMDDataType** member MUST be set to one of the following values.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.

Value	Meaning
BINARY_METADATA 0x00000003	Specifies binary data in any form.
DWORD_METADATA 0x00000001	Specifies all **DWORD** (unsigned 32-bit integer) data.
EXPANDSZ_METADATA 0x00000004	Specifies all data that consists of a string that includes the NULL-terminating character, and which contains environment variables that are not expanded.
MULTISZ_METADATA 0x00000005	Specifies all data represented as an array of strings, where each string includes the NULL-terminating character, and the array itself is terminated by two NULL-terminating characters.
STRING_METADATA 0x00000002	Specifies all data consisting of an ASCII string that includes the NULL-terminating character.

dwMDDataLen: An unsigned integer value that specifies the length of the data in bytes. If the data is a string, this value includes the NULL-terminating character. For lists of strings, this includes an additional NULL-terminating character after the final string (double NULL-terminating characters).

For example, the length of a string list containing two strings would be:

```
(wcslen(stringA) + 1) * sizeof(WCHAR) + (wcslen(stringB) + 1)
 * sizeof(WCHAR) + 1 * sizeof(WCHAR)
```

In-process clients need to specify **dwMDDataLen** only when setting binary data in the metabase. Remote clients MUST specify **dwMDDataLen** for all data types.

pbMDData: When setting data in the metabase, this member contains a pointer to a buffer that holds the data. When getting data from the metabase, this member contains a pointer to a buffer that will receive the data.

dwMDDataTag: A reserved member that is currently unused.

2.2.7 MD_CHANGE_OBJECT_W

The **MD_CHANGE_OBJECT_W** structure defines information about changed configuration data; this data is being sent to the registered change notification clients.

```
typedef struct {
  [string] WCHAR* pszMDPath;
  DWORD dwMDChangeType;
  DWORD dwMDNumDataIDs;
  [unique, size_is(dwMDNumDataIDs)]
    DWORD* pdwMDDataIDs;
} MD_CHANGE_OBJECT_W;
```

pszMDPath: A pointer to a Unicode string that contains the changed key path.

dwMDChangeType: An unsigned integer value specifying the code of change operation that happened with the key. The **dwMDChangeType** member MUST be set to one of the following values:

Value	Meaning
MD_CHANGE_TYPE_DELETE_OBJECT 0x00000001	The Meta Object was deleted.
MD_CHANGE_TYPE_ADD_OBJECT 0x00000002	The Meta Object was added.
MD_CHANGE_TYPE_SET_DATA 0x00000004	A data item was set.
MD_CHANGE_TYPE_DELETE_DATA 0x00000008	A data item was deleted.
MD_CHANGE_TYPE_RENAME_OBJECT 0x00000010	The Meta Object was renamed.
MD_CHANGE_TYPE_RESTORE 0x00000020	The metabase was restored.

dwMDNumDataIDs: A unsigned integer value specifying a size of array containing configuration data IDs that were changed.

pdwMDDataIDs: A pointer to an integer array containing data IDs of changed key properties.

2.2.8 METADATA_MASTER_ROOT_HANDLE

This predefined handle points to the root of configuration storage tree. It is represented by a NULL handle and declared in the following way.

```
#define METADATA_MASTER_ROOT_HANDLE     0
```

3 Protocol Details

The following sections specify details of the IIS IMSAdminBaseW Remote Protocol, including abstract data models, interface method syntax, and message processing rules.

3.1 IMSAdminBaseW Server Role Details

3.1.1 Abstract Data Model

The following information MUST be maintained by the server for use in responding to client queries and commands.

Configuration storage, interfaced by **IMSAdminBaseW** SHOULD be implemented as a hierarchical tree-like store of data, the same type as Windows registry, but with important extensions. Configuration data is accessed through the metabase path, where each node of the path represents branch of the tree, similar to the registry key. The node is identified by name that should be unique between siblings and the metabase path is combined from node names separated by predefined separation characters. Each node could contain any number of data value items (data) identified by numerical IDs, and any number of child nodes.

In addition to the registry-like features, the metabase provides data value items inheritance along the metabase path in such a manner, that data value item defined on the node located closer to the root of the tree could be inherited by lower level nodes. Each data value item carries an attribute that could be used to find, if the data on any particular node is defined on that node, or inherited from the parent node.

Each data on the metabase node has attributes describing the type of data that it contains and type of use for this data. For a complete description of the data structure with all the attributes, see **METADATA_RECORD**.

The Metabase root is defined by predefined handle METADATA_MASTER_ROOT_HANDLE. When the metabase is initialized, this handle is opened with read access and stays opened during the entire session. When a caller is getting access to the nodes, which are located lower than root, the access type should be passed as parameter. This access type could be read or write, see **OpenKey**. When a caller requests write access, the server locks the metabase sub-tree starting from the node where access is requested, including the parental nodes and all the child nodes. If at the moment of call, the requested part of metabase is already locked by another caller, then the requesting call returns Win32 error code ERROR_PATH_BUSY (see [MS-ERREF], section 3). The server MUST keep the state of the locked subtree until the opened node will be explicitly closed. When the caller requests read-only access, the server locks the same portion of the tree from being opened for write access. Multiple calls could open locked nodes for read-only access at the same time. If any caller requests write access to the portion of the tree, which is currently locked for read-only access, then this call will return the Win32 error code ERROR_PATH_BUSY (see [MS-ERREF], section 3).

The server MUST keep the counter of changes that were done to the configuration storage.

The server MUST keep record of last change time for each node.

3.1.1.1 Secure Session Context

When the client expects to exchange sensitive data marked with the METADATA_SECURE secure flag, it will negotiate secure session. As part of the secure session negotiation, both client and server will build secure session context.

For each client the server MUST maintain the following information related to secure session:

- The server's **Key Exchange Private** and Public Key.
- The server's **Signature Private** and Public Key.
- The client's Key Exchange Public Key.
- The client's Signature Public Key.
- The server's Session Key.
- The client's Session Key.

3.1.2 Timers

No protocol timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3.2.3.2.1.

3.1.3 Initialization

The IIS IMSAdminBaseW Remote Protocol server MUST be initialized by registering the RPC interface and listening on the RPC **well-known endpoint**, as specified in section 2.1. The server MUST then wait for IIS IMSAdminBaseW Remote Protocol clients to establish a connection.

3.1.4 Message Processing Events and Sequencing Rules

This DCOM interface inherits the IUnknown interface. Method **opnum** field values start with 3; opnum values 0 through 2 represent the IUnknown_QueryInterface, IUnknown_AddRef, and IUnknown_Release methods, respectively, as specified in [MS-DCOM].

Methods with opnum field values 34 through 39 are defined in section 3.3.4, and field value 40 is defined in section 3.5.4.

This protocol MUST indicate to the RPC runtime that it is to perform a strict **Network Data Representation (NDR)** data consistency check at target level 5.0, as specified in [MS-RPCE] section 3.

Methods in RPC Opnum Order

Method	Description
AddKey	Creates a node and adds it to the metabase as a subnode of an existing node at the specified path. Opnum: 3
DeleteKey	Deletes a node and all its data from the metabase. All of the node's subnodes are recursively deleted. Opnum: 4
DeleteChildKeys	Deletes all subnodes of the specified node and any data they contain. It also recursively deletes all nodes below the subnodes. Opnum: 5
EnumKeys	Enumerates the subnodes of the specified node. Opnum: 6
CopyKey	Copies or moves a node, including its subnodes and data, to a specified

Method	Description
	destination. The copied or moved node becomes a subnode of the destination node. Opnum: 7
RenameKey	Renames a node in the metabase. Opnum: 8
R_SetData	Sets a data item for a particular node in the metabase. Opnum: 9
R_GetData	Returns a data entry from a particular node in the metabase. Opnum: 10
DeleteData	Deletes specific data entries from a node in the metabase. Opnum: 11
R_EnumData	Enumerates the data entries of a node in the metabase. Opnum: 12
R_GetAllData	Returns all data associated with a node in the metabase, including all values that the node inherits. Opnum: 13
DeleteAllData	Deletes all or a subset of local data associated with a particular node. Opnum: 14
CopyData	Copies or moves data between nodes. Opnum: 15
GetDataPaths	Returns the paths of all nodes in the subtree relative to a specified starting node that contains the supplied identifier. Opnum: 16
OpenKey	Opens a node for read access, write access, or both. The returned handle can be used by several of the other methods in **IMSAdminBaseW**. Opnum: 17
CloseKey	Closes a handle to a node. Opnum: 18
ChangePermissions	Changes permissions on an open handle. Opnum: 19
SaveData	Explicitly saves the metabase data to disk. Opnum: 20
GetHandleInfo	Returns information associated with the specified metabase handle. Opnum: 21
GetSystemChangeNumber	Returns the number of changes made to data since the metabase was created. Opnum: 22

Method	Description
GetDataSetNumber	Returns all the data set numbers associated with a node in the metabase. Opnum: 23
SetLastChangeTime	Sets the last change time associated with a node in the metabase. Opnum: 24
GetLastChangeTime	Returns the last change time associated with a node in the metabase. Opnum: 25
R_KeyExchangePhase1	Receives a pair of encrypted client nodes and returns server encryption and session keys. Opnum: 26
R_KeyExchangePhase2	Receives the encrypted client session and hash keys in response to **R_KeyExchangePhase1** and returns the encrypted server hash keys. Opnum: 27
Backup	Backs up the metabase to a specified location. Opnum: 28
Restore	Restores the metabase from a backup. Opnum: 29
EnumBackups	Enumerates the metabase backups in a specified backup location, or in all backup locations. Opnum: 30
DeleteBackup	Deletes a metabase backup from a backup location. Opnum: 31
UnmarshalInterface	Unmarshals a reference to the **IMSAdminBaseW** interface. Opnum: 32
R_GetServerGuid	Returns the **GUID** for the IIS instance that is running. Opnum: 33

Structures

The **Message Processing Events and Sequencing Rules** interface defines the following structures.

Structure	Description
METADATA_HANDLE_INFO	Defines information about a handle to a metabase entry.
METADATA_RECORD	Defines information about a metabase entry.
METADATA_GETALL_RECORD	Defines an analogous structure to **METADATA_RECORD**, but is used only to return data from a call to the **R_GetAllData** method.
IIS_CRYPTO_BLOB	Defines a block of opaque data, possibly encrypted, for RPC marshaling between IIS and a client.

3.1.4.1 Transferring Sensitive Data

Some of the data that is transferred between client and server is of a sensitive nature and needs to be protected. An example of sensitive data is a password. The IMSAdminBaseW Remote protocol defines a way to protect sensitive data transferred in the **METADATA_RECORD** or **METADATA_GETALL_RECORD** structures.

When the client expects transfer of sensitive data, it will initiate negotiation of a secure session. The secure session is negotiated by processing **R_KeyExchangePhase1** and **R_KeyExchangePhase2** calls(). 512-bit RSA Key Exchange Keys are used to exchange 40-bit **RC4** session keys. RC4 Session Keys (one for client and one for server) are used to encrypt data over the wire. An **MD5 hash** signed with 512-bit RSA Signature Keys is used for message integrity checks.<1>

There are four methods that take advantage of this protection:

- **R_GetData**
- **R_EnumData**
- **R_GetAllData**
- **R_SetData**

Sensitive data is marked with the METADATA_SECURE secure flag in the **METADATA_RECORD** or **METADATA_GETALL_RECORD** structure.<2>

3.1.4.1.1 Secure Session Negotiation Server Role

The purpose of the secure session negotiation is to exchange Session Keys and Signature Public Keys between the server and client. The Session Keys are used for encryption and **decryption** of sensitive data, and Signature Public Keys are used to ensure message integrity.

Secure session negotiation is initiated by the client using the **R_KeyExchangePhase1** and **R_KeyExchangePhase2** call sequence; for more information, see 3.2.4.1. The server participates in the secure session negotiation by responding to **R_KeyExchangePhase1** and **R_KeyExchangePhase2** calls, in that order.

The server MUST participate in the secure session negotiation initiated by the client. As a result of the secure session negotiation, the server will receive the client's Session Key and Signature Public Key.

3.1.4.1.2 Encrypting Data

Some data transferred between the client and server must be encrypted before it is sent. Encrypted data will be transferred in an **IIS_CRYPTO_BLOB** message with the **BlobSignature** field set to ENCRYPTED_DATA_BLOB_SIGNATURE.

Secure session MUST be negotiated before the data encryption takes place (see section 3.1.4.1.1).

Sender MUST perform the following steps to encrypt data and build **IIS_CRYPTO_BLOB**:

1. Create an instance of a CLEARTEXT_WITH_PREFIX_BLOB message.
   - Set the **Reserved** field to zero.
   - Place the data to be encrypted into the **ClearTextData** field.

2. Calculate the signed hash and hash length of the CLEARTEXT_WITH_PREFIX_BLOB message from the previous step, as specified in section 3.1.4.1.4.

3. Encrypt the CLEARTEXT_WITH_PREFIX_BLOB message data using the Session Key of the receiver. Client will use Server's Session Key and server will use Client's Session Key.

4. Create an instance of ENCRYPTED_DATA_BLOB.

   - Set **EncryptedDataLength** to the number of encrypted bytes from the previous step.
   - Store encrypted data from the earlier step in the **EncryptedData** field.
   - Calculate the padding size between zero and seven, so that **EncryptedDataLength** + padding length is a multiple of eight. Set padding bytes to 0x00.
   - Set the **SignedHashLength** and **SignedHash** calculated in earlier step.

5. Create an instance of an **IIS_CRYPTO_BLOB** message.

   - Set **BlobSignature** to ENCRYPTED_DATA_BLOB_SIGNATURE.
   - Calculate the **BlobDataLength** field value in the **IIS_CRYPTO_BLOB** message by adding the **EncryptedDataLength** + padding length + **SignedHashLength**.
   - Store the ENCRYPTED_DATA_BLOB instance from the earlier step in the **BlobData** field.

3.1.4.1.3 Decrypting Data

Some data is encrypted before it is transferred between the client and server. The receiver MUST decrypt the data before it can be used. Encrypted data is stored in **IIS_CRYPTO_BLOB** message with the **BlobSignature** field set toENCRYPTED_DATA_BLOB_SIGNATURE.

The data decryption process assumes that secure session was already negotiated (see section 3.1.4.1.1).

The receiver MUST perform the following steps to decrypt the data:

1. Retrieve the **BlobData** field from an **IIS_CRYPTO_BLOB** message.

2. Interpret **BlobData** as ENCRYPTED_DATA_BLOB message.

3. Retrieve the **EncryptedData** field or **EncryptedDataLength** bytes from the ENCRYPTED_DATA_BLOB message.

4. Decrypt the **EncryptedData** data using the session key of the receiver. The server will use server's Session Key and the client will use client's Session Key.

5. Follow instructions in section 3.1.4.1.5 to validate the hash. Use decrypted data from the previous step.

6. Interpret the decrypted data from the earlier step as CLEARTEXT_WITH_PREFIX_BLOB message.

7. Retrieve the **ClearTextData** field from the CLEARTEXT_WITH_PREFIX_BLOB message. It will contain the final decrypted data.

3.1.4.1.4 Signed Hash Calculation

The signed hash is used to provide integrity checking by the receiver.

The sender MUST perform the following steps to calculate the hash:

1. Compute an MD5 hash of cleartext data.
2. Use the sender's Signature Private Key (the server will use the server's Signature Private Key and the client will use the client's Signature Private Key) to sign the MD5 hash, as specified in [RFC3447].
3. Size of the hash will match the number of bits in the signature key. 512-bit RSA Signature Keys will be used for signing, so the signed hash will always be 0x40 bytes long.

3.1.4.1.5 Signed Hash Validation

Validation is to be performed by the receiver to verify the integrity of the received data.

The following steps MUST be performed by the receiver.

1. Compute an MD5 hash of decrypted data.
2. Use the MD5 hash from previous step and the sender's Signature Public Key to verify against the **SignedHash** field stored in the **IIS_CRYPTO_BLOB** message. The server will use the client's Signature Public Key and the client will use the server's Signature Public Key for verification. If the signature does not match, then the validation fails, as specified in [RFC3447].

3.1.4.2 OpenKey (Opnum 17)

The **OpenKey** method opens a node for read access, write access, or both. The returned handle can be used by several of the other methods in **IMSAdminBaseW**.

```
HRESULT OpenKey(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] DWORD dwMDAccessRequested,
  [in] DWORD dwMDTimeOut,
  [out] METADATA_HANDLE* phMDNewHandle
);
```

hMDHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node to be opened, relative to *hMDHandle*.

dwMDAccessRequested: A set of bit flags specifying the requested permissions for the handle. This parameter MUST be set to at least one of the following values.

Value	Meaning
METADATA_PERMISSION_READ 0x00000001	Open the node for reading.
METADATA_PERMISSION_WRITE 0x00000002	Open the node for writing.

dwMDTimeOut: An unsigned 32-bit integer value specifying the time, in milliseconds, for the method to wait on a successful open operation.

phMDNewHandle: A pointer to the newly opened metadata handle (**DWORD**).

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000003 ERROR_PATH_NOT_FOUND	The system cannot find the path specified.
0x00000006 ERROR_INVALID_HANDLE	The handle is invalid.
0x00000057 ERROR_INVALID_PARAMETER	The parameter is incorrect.
0x00000094 ERROR_PATH_BUSY	The path specified cannot be used at this time.
0x000006A6 RPC_S_INVALID_BINDING	The binding handle is invalid.
0x000006BD RPC_S_NO_CALL_ACTIVE	There are no Remote Procedure Call (RPC) s active on this thread.
0x000006E4 RPC_S_CANNOT_SUPPORT	The requested operation is not supported.
0x80000003 E_INVALIDARG	One or more arguments are invalid.
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.
0x800CC800 MD_ERROR_NOT_INITIALIZED	Metadata has not been initialized.

The **opnum** field value for this method is 17.

When processing this call the server MUST do the following:

- Check that the handle parameter passed as the parent handle. This handle could be either a root handle or a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check that the relative path points to a valid node; otherwise, return an error.
- Determine if it is possible to provide the required access type for the destination node with the path combined from the parent handle path and the relative path.

- If the destination node represents the root of the metabase and the requested access is for write, then the server returns an error.
- If the destination node falls into part of the metabase that is locked for write access, then the server will attempt to provide access during timeout, which is passed as a parameter. If, after this timeout, the node is still locked, then the server returns an error code. The server MAY poll the state of the metabase and continue processing before the timeout is expired if the node will be unlocked.
- If write access is requested and the destination node falls into part of the metabase that is locked for read-only access, then the server will attempt to provide write access during timeout which is passed as a parameter. If, after this timeout, the node is still locked, then the server returns an error code. The server MAY poll the state of the metabase and continue processing before the timeout is expired, if the node will be unlocked.
- If access could be provided, the server calculates the handle of the destination node, increases its lock count and saves its state.

Return the following information to the client:

- The handle of opened node.

3.1.4.3 CloseKey (Opnum 18)

The **CloseKey** method closes a handle to a node.

```
HRESULT CloseKey(
  [in] METADATA_HANDLE hMDHandle
);
```

hMDHandle: An unsigned 32-bit integer value containing the handle to close, as returned by the **OpenKey** method.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function, CoInitialize, has not been called. For more information, see [MSDN-CoInitialize].

The opnum field value for this method is 18.

If the handle was opened with write permission and changes have been made, this will cause any registered callback methods to be called.

When processing this call the server MUST do the following:

- Check the handle parameter passed as a parent handle. This handle should be a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Decrease the internal lock count in the state of the handle and release the lock, if it is possible.

3.1.4.4 AddKey (Opnum 3)

The **AddKey** method creates a node and adds it to the metabase as a subnode of an existing node at the specified path.

```
HRESULT AddKey(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath
);
```

hMDHandle: An unsigned 32-bit integer value that specifies a handle to a node in the metabase with write permissions as returned by the **OpenKey**. This handle SHOULD not be the master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the new node's path, relative to the path of *hMDHandle*.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function, CoInitialize, has not been called. For more information, see [MSDN-CoInitialize].
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.

The opnum field value for this method is 3.

When processing this call the server MUST do the following:

- Check the handle parameter. This handle SHOULD be the root handle, or a handle returned from the **OpenKey** call. If handle is invalid, return an error.
- Check if parent handle was opened for write access. Otherwise, return an error.
- Check if relative path has right syntax and length. If not, return an error.
- Add a new node to the tree that has the result path as a combined path of parent node and relative path. If any intermediate nodes required, the server creates these nodes.

3.1.4.5 CopyKey (Opnum 7)

The **CopyKey** method copies or moves a node, including its subnodes and data, to a specified destination. The copied or moved node becomes a subnode of the destination node.

```
HRESULT CopyKey(
  [in] METADATA_HANDLE hMDSourceHandle,
  [unique, in, string] LPCWSTR pszMDSourcePath,
  [in] METADATA_HANDLE hMDDestHandle,
  [unique, in, string] LPCWSTR pszMDDestPath,
  [in] BOOL bMDOverwriteFlag,
  [in] BOOL bMDCopyFlag
);
```

hMDSourceHandle: An unsigned 32-bit integer value containing the handle of the node in the metabase to be copied or moved. If the node is to be copied (*hMDCopyFlag* is set to TRUE), *hMDSourceHandle* MUST be opened with read permissions. If the node is to be moved (*hMDCopyFlag* is set to FALSE), *hMDSourceHandle* MUST be opened with read/write permissions.

pszMDSourcePath: A pointer to a Unicode string that contains the path of the node to be copied or moved relative to the path of *hMDSourceHandle*.

hMDDestHandle: A handle opened with write permissions that specifies the destination of the moved or copied node.

pszMDDestPath: A pointer to a string that contains the path of the new or moved node, relative to *hMDDestHandle*.

bMDOverwriteFlag: A Boolean value that determines the behavior if the destination node already exists. If TRUE, the existing node and all its data and children are deleted prior to copying or moving the source. If FALSE, the existing node, data, and children remain, and the source is merged with that data. In cases of data conflicts, the source data overwrites the destination data.

bMDCopyFlag: A Boolean value that specifies whether to copy or move the specified node. If TRUE, the node is copied. If FALSE, the node is moved, and the source node is deleted from its original location.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function CoInitialize has not been called. For more information, see [MSDN-CoInitialize].
0x8007000E	There was not enough memory to complete the method call.

Return value/code	Description
E_OUTOFMEMORY	

The opnum field value for this method is 7.

When processing this call the server MUST do the following:

- Check the source handle parameter. This handle SHOULD be the root handle or a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check the destination handle parameter. This handle SHOULD be the root handle or a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if the destination handle was opened for write access. Otherwise, return an error.
- Check if the source relative path points to the existing node. Otherwise, return an error.
- Check if destination relative path has the right syntax and length. If not, return an error.
- Check if the destination node exists. If it is true, then check if the overwrite parameter if set to TRUE. If it is FALSE, return an error.
- If the destination node does not exist, add a new node to the tree that has the result path as a combined path of destination parent node and destination relative path. If any intermediate nodes are required, the server creates these nodes.
- Copy all data from the source path to the destination path.
- If the copy flag is set to FALSE, delete the source node.

3.1.4.6 DeleteKey (Opnum 4)

The **DeleteKey** method deletes a node and all its data from the metabase. All of the subnodes are recursively deleted.

```
HRESULT DeleteKey(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath
);
```

hMDHandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with write permissions, as returned by the **OpenKey**.

pszMDPath: A pointer to a Unicode string that contains the path of the node to be deleted, relative to the path of *hMDHandle*. This parameter MUST NOT be NULL.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function CoInitialize has not been called. For more information, see [MSDN-CoInitialize].
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.

The opnum field value for this method is 4.

When processing this call the server MUST do the following:

- Check that the handle parameter is passed as the parent handle. This handle SHOULD be a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if a handle was opened for write access. Otherwise, return an error.
- Check if the relative path points to the existing subnode of parent handle. If not, return an error.
- Delete the node that contains the path which was calculated to be the path of the parent handle combined with the relative path.
- Delete all child nodes of this node.

3.1.4.7 DeleteChildKeys (Opnum 5)

The **DeleteChildKeys** method deletes all subnodes of the specified node and any data they contain. It also recursively deletes all nodes below the subnodes.

```
HRESULT DeleteChildKeys(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath
);
```

hMDHandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with write permissions, as returned by the **OpenKey** method.

pszMDPath: A pointer to a Unicode string that contains the path of the node whose subnodes are to be deleted, relative to the path of the *hMDHandle* parameter.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

Return value/code	Description
0x00000003 ERROR_PATH_NOT_FOUND	The system cannot find the path specified.
0x00000005 ERROR_ACCESS_DENIED	Access is denied.
0x800CC800 MD_ERROR_NOT_INITIALIZED	Metadata has not been initialized.

The opnum field value for this method is 5.

When processing this call the server MUST do the following:

- Check that the handle parameter is passed as the parent handle. This handle SHOULD be a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if the handle was opened for write access. Otherwise, return an error.
- Check if the relative path points to the existing subnode of the parent handle. Otherwise, return an error.
- Delete all child nodes of this subnode.

3.1.4.8 DeleteData (Opnum 11)

The **DeleteData** method deletes specific data entries from a node in the metabase.

```
HRESULT DeleteData(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] DWORD dwMDIdentifier,
  [in] DWORD dwMDDataType
);
```

hMDHandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with write permissions as returned by the **OpenKey**. This handle MUST NOT be the master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node whose subnodes are to be deleted, relative to the path of the *hMDHandle* parameter.

dwMDIdentifier: An integer value specifying the data identifier.

dwMDDataType: An integer value specifying a data type. If this parameter is not set to ALL_METADATA, the data item will be removed only if its data type matches the specified type.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.
BINARY_METADATA	Specifies binary data in any form.

Value	Meaning
0x00000003	
DWORD_METADATA 0x00000001	Specifies all **DWORD** (unsigned 32-bit integer) data.
EXPANDSZ_METADATA 0x00000004	Specifies all data consisting of a string that includes the NULL-terminating character, which contains unexpanded environment variables.
MULTISZ_METADATA 0x00000005	Specifies all data represented as an array of strings, where each string contains two occurrences of the NULL-terminating character.
STRING_METADATA 0x00000002	Specifies all data consisting of an ASCII string that includes the NULL-terminating character.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function, CoInitialize, has not been called. For more information, see [MSDN-CoInitialize].
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.

The opnum field value for this method is 11.

When processing this call the server MUST do the following:

- Check that the handle parameter is passed as the parent handle. This handle SHOULD be a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if the handle was opened for write access. Otherwise, return an error.
- Check if the relative path points to the existing subnode of the parent handle. Otherwise, return an error.
- Check if the node has data with an ID equal to the ID parameter passed from the client.
- Check the data type parameter. If it has bits set for all data types, delete this data from the node.
- If the data type parameter points to a particular data type, and if its data type is the same as the passed parameter, delete the data; otherwise, return an error.

3.1.4.9 DeleteAllData (Opnum 14)

The **DeleteAllData** method deletes all or a subset of local data associated with a particular node.

```
HRESULT DeleteAllData(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] DWORD dwMDUserType,
  [in] DWORD dwMDDataType
);
```

hMDHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase with write permissions as returned by the **OpenKey**. This handle MUST NOT be the master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node with which the data to be deleted is associated, relative to the path of *hMDHandle*.

dwMDUserType: An integer value specifying the data to delete based on user type.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.
ASP_MD_UT_APP 0x00000065	Specifies data specific to ASP application configuration.
IIS_MD_UT_FILE 0x00000002	Specifies data specific to a file, such as access permissions or logon methods.
IIS_MD_UT_SERVER 0x00000001	Specifies data specific to the server, such as ports in use and IP addresses.
IIS_MD_UT_WAM 0x00000064	Specifies data specific to Web application management.

dwMDDataType: An integer value specifying a data type. If this parameter is not set to ALL_METADATA, the data item will be removed only if its data type matches the specified type.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.
BINARY_METADATA 0x00000003	Specifies binary data in any form.
DWORD_METADATA 0x00000001	Specifies all **DWORD** (unsigned 32-bit integer) data.
EXPANDSZ_METADATA 0x00000004	Specifies all data consisting of a string that includes the NULL terminating character, which contains unexpanded environment variables.
MULTISZ_METADATA 0x00000005	Specifies all data represented as an array of strings, where each string contains two occurrences of the NULL terminating character.
STRING_METADATA 0x00000002	Specifies all data consisting of an ASCII string that includes the NULL terminating character.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 14.

When processing this call the server MUST do the following:

- Check that the handle parameter is passed as a parent handle. This handle SHOULD be a handle that is returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if the handle was opened for write access. Otherwise return an error.
- Check if the relative path points to the existing subnode of the parent handle. Otherwise, return an error.
- For each data value that is defined on the destination node and not inherited from the parent node, repeat the following:
- If the user type parameter and data type parameter are equal to "all data", delete the data.
- If the user type parameter equals the user type of the data and the data type parameter is equal to the data type of the data, delete this data value.

3.1.4.10 CopyData (Opnum 15)

The **CopyData** method copies or moves data between nodes.

```
HRESULT CopyData(
  [in] METADATA_HANDLE hMDSourceHandle,
  [unique, in, string] LPCWSTR pszMDSourcePath,
  [in] METADATA_HANDLE hMDDestHandle,
  [unique, in, string] LPCWSTR pszMDDestPath,
  [in] DWORD dwMDAttributes,
  [in] DWORD dwMDUserType,
  [in] DWORD dwMDDataType,
  [in] BOOL bMDCopyFlag
);
```

hMDSourceHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDSourcePath: A pointer to a Unicode string that contains the path of the node with which the source data is associated, relative to the path of *hMDSourceHandle*.

hMDDestHandle: The handle of the node to which the data will be copied, with write permissions.

pszMDDestPath: A pointer to a Unicode string that contains the path of the node for data to be copied to, relative to the path of *hMDDestHandle*.

dwMDAttributes: Flags used to filter the data, as specified in the **METADATA_RECORD** structure.

dwMDUserType: An integer value specifying the data to copy based on the user type.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of user type.
ASP_MD_UT_APP 0x00000065	Specifies data specific to ASP application configuration.
IIS_MD_UT_FILE 0x00000002	Specifies data specific to a file, such as access permissions or logon methods.
IIS_MD_UT_SERVER 0x00000001	Specifies data specific to the server, such as ports in use and IP addresses.
IIS_MD_UT_WAM 0x00000064	Specifies data specific to Web application management.

dwMDDataType: An integer value specifying a data type. If this parameter is not set to ALL_METADATA, the data item will be copied only if its data type matches the specified type.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.
BINARY_METADATA 0x00000003	Specifies binary data in any form.
DWORD_METADATA 0x00000001	Specifies all **DWORD** (unsigned 32-bit integer) data.
EXPANDSZ_METADATA 0x00000004	Specifies all data consisting of a string that includes the NULL-terminating character, which contains unexpanded environment variables.
MULTISZ_METADATA 0x00000005	Specifies all data represented as an array of strings, where each string contains two occurrences of the NULL-terminating character.
STRING_METADATA 0x00000002	Specifies all data consisting of an ASCII string that includes the NULL-terminating character.

bMDCopyFlag: A Boolean value that specifies whether to copy or move the data. If this parameter is set to TRUE, the data is copied. If it is FALSE, the data is moved.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the

method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function, CoInitialize, has not been called. For more information, see [MSDN-CoInitialize].
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.

The opnum field value for this method is 15.

When processing this call the server MUST do the following:

- Check the *hMDSourceHandle* parameter. This handle SHOULD be the root handle, or a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check the *hMDDestHandle* parameter. This handle SHOULD be the root handle, or a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if the destination handle was opened for write access. Otherwise, return an error.
- Check if the source relative path points to the existing node. Otherwise, return an error.
- Check if the destination relative path points to an existing node. If not, return an error.
- If the *dwMDUserType* and the *dwMDDataType* parameters are not equal to "all metadata", then use these parameters as the data selection filter.
- If the *dwMDAttributes* parameter is defined, use this parameter to get the data.
- Copy the selected data from the source node to the destination node.
- If the *bMDCopyFlag* parameter is set to false, remove the selected data from the source.

3.1.4.11 EnumKeys (Opnum 6)

The **EnumKeys** method enumerates the subnodes of the specified node.

```
HRESULT EnumKeys(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [out, size_is(ADMINDATA_MAX_NAME_LEN)]
    LPWSTR pszMDName,
  [in] DWORD dwMDEnumObjectIndex
);
```

hMDHandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node whose subnodes are to be enumerated, relative to the path of *hMDHandle*.

pszMDName: A pointer to a string buffer that receives the names of the enumerated metabase subnodes.

dwMDEnumObjectIndex: An integer value specifying the index of the subnode to be retrieved.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function, CoInitialize, has not been called. For more information, see [MSDN-CoInitialize].
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.

The **opnum** field value for this method is 6.

A subnode can be enumerated once per call. Subnodes are numbered from zero to (NumKeys - 1), with NumKeys equal to the number of subnodes below the node.

When processing this call the server MUST do the following:

- Check that the handle parameter is passed as the parent handle. This handle SHOULD be a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if the handle was opened for read access. Otherwise, return an error.
- Check if the relative path points to the existing subnode of the parent handle. Otherwise, return an error.
- Check that the string pointer that is passed for the name is not NULL and points to the memory block that is at least METADATA_MAX_NAME_LEN size.
- Find the child node of the destination node that has an index equal to the parameter *dwMDEnumKeyIndex*. If there is no child with a passed index, return the error ERROR_NO_MORE_ITEMS.

Copy the name of this child node to the name buffer.

3.1.4.12 R_EnumData (Opnum 12)

The **R_EnumData** method enumerates the data entries of a node in the metabase.

```
HRESULT R_EnumData(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
```

```
  [in, out] METADATA_RECORD* pmdrMDData,
  [in] DWORD dwMDEnumDataIndex,
  [out] DWORD* pdwMDRequiredDataLen,
  [out] IIS_CRYPTO_BLOB** ppDataBlob
);
```

hMDHandle: An unsigned 32-bit integer value containing the handle to a node in the metabase. The handle can be the metadata master root handle (0x00000000) or a handle, with read permission, returned by the **OpenKey** method.

pszMDPath: A pointer to a Unicode string that contains the path of the node to be enumerated, relative to the path of *hMDHandle*.

pmdrMDData: A pointer to a **METADATA_RECORD** structure that specifies the retrieved data.

dwMDEnumDataIndex: An integer value specifying the index of the entry retrieved.

pdwMDRequiredDataLen: Pointer to a **DWORD** that receives the required buffer size if the method returns ERROR_INSUFFICIENT_BUFFER.

ppDataBlob: An **IIS_CRYPTO_BLOB** structure containing the requested values as encrypted opaque data.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 12.

When processing this call the server MUST do the following:

- Check that the handle parameter is passed as the parent handle. This handle SHOULD be a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
- Check if the handle was opened for read access. Otherwise, return an error.
- Check if relative path points to the existing subnode of the parent handle. Otherwise, return an error.
- Check if the pointer to the required data length parameter is not NULL. Otherwise, return an error.
- Obtain the requested data using an index parameter. If the index points to non-existing data, return the error ERROR_NO_MORE_ITEMS.
- Check the pointer to **METADATA_RECORD**. If it is NULL, the server will calculate the exact size of the buffer that is required for the requested data. Copy this data to the required data length parameter, and return the error code ERROR_INSUFFICIENT_BUFFER.

- Copy the requested data to the **METADATA_RECORD** buffer.
- The server SHOULD handle parameters **IIS_CRYPTO_BLOB** and **METADATA_RECORD** identically to the rules in **R_GetData**.

3.1.4.13 Backup (Opnum 28)

The **Backup** method backs up the metabase to a specified location.

```
HRESULT Backup(
  [unique, in, string] LPCWSTR pszMDBackupLocation,
  [in] DWORD dwMDVersion,
  [in] DWORD dwMDFlags
);
```

pszMDBackupLocation: A string of up to 100 Unicode characters that identifies the backup location containing the backup to be restored.

dwMDVersion: An integer value specifying the version number to be used for the backup.

Value	Meaning
MD_BACKUP_HIGHEST_VERSION 0xFFFFFFFE	Overwrite the highest existing backup version in the specified backup location.
MD_BACKUP_NEXT_VERSION 0xFFFFFFFF	Use the next backup version number available in the specified backup location.

dwMDFlags: An integer value containing the bit flags describing the type of backup operation to be performed. The flags can be one or more of the following values.

Value	Meaning
MD_BACKUP_FORCE_BACKUP 0x00000004	Force the backup even if the **SaveData** operation specified by MD_BACKUP_SAVE_FIRST fails.
MD_BACKUP_OVERWRITE 0x00000001	Back up even if a backup of the same name and version exists in the specified backup location, overwriting it if necessary.
MD_BACKUP_SAVE_FIRST 0x00000002	Perform a **SaveData** operation before the backup.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 28.

The location string can be up to 100 Unicode characters in length. Multiple metabase backups can be stored in a single backup location.

When processing this call the server MUST do the following:

- Check the parameter backup location. It SHOULD point to the existing place in the configuration store or the file system. The exact meaning of this is implementation-dependent. Return an error if the location does not exist.
- Check the version parameter. If it is greater than the maximum allowed version number and is not either the MD_BACKUP_HIGHEST_VERSION or the MD_BACKUP_NEXT_VERSION version numbers, the server returns the E_INVALIDARG error code.
- If the parameter flags have the bit MD_BACKUP_SAVE_FIRST set, flush the in-memory configuration data first. If this operation fails, check the MD_BACKUP_FORCE_BACKUP bit. If this bit is reset, return an error. Otherwise, continue the operation.
- Check the MD_BACKUP_OVERWRITE bit. If it is reset, check if a backup with the target version exists. If it is TRUE, return an error, otherwise overwrite the existing backup.
- If the backup location is an empty string, the server backs up the configuration to the default backup location.
- The server saves the persisted data using the backup location and the version number as a key so that the data can be restored later.

3.1.4.14 EnumBackups (Opnum 30)

The **EnumBackups** method enumerates the metabase backups in a specified backup location, or in all backup locations.

```
HRESULT EnumBackups(
  [in, out, size_is(MD_BACKUP_MAX_LEN)]
    LPWSTR pszMDBackupLocation,
  [out] DWORD* pdwMDVersion,
  [out] PFILETIME pftMDBackupTime,
  [in] DWORD dwMDEnumIndex
);
```

pszMDBackupLocation: A string of Unicode characters containing an empty string or a string of up to 100 Unicode characters that identifies the backup location.

pdwMDVersion: An integer value containing the version number of the backup.

pftMDBackupTime: A **FILETIME** structure containing the Coordinated Universal Time (UTC) date and time when this backup was created.

dwMDEnumIndex: An integer value specifying the index number of the backup to be enumerated.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the

method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 30.

When processing this call the server MUST do the following:

- Check the location parameter. This parameter SHOULD point to a string buffer that is at least 256 characters long. Otherwise, return an error.
- If the *pszMDBackupLocation* parameter is not an empty string, check the location defined by this parameter. If the location does not exist, return an error.
- If the *pszMDBackupLocation* parameter is an empty string, the server SHOULD enumerate backups in all known locations; otherwise it will do it for the requested location only.
- For the current location, find the backup with an index equal to the index parameter. If it does not exist, return the error ERROR_NO_MORE_ITEMS.
- Copy a version of the backup into the *pdwMDVersion* parameter.
- Copy the backup time into the *pftMDBackupTime* parameter.
- Copy the backup location to the location parameter buffer.

3.1.4.15 DeleteBackup (Opnum 31)

The **DeleteBackup** method deletes a metabase backup from a backup location.

```
HRESULT DeleteBackup(
  [unique, in, string] LPCWSTR pszMDBackupLocation,
  [in] DWORD dwMDVersion
);
```

pszMDBackupLocation: A string of up to 100 Unicode characters that identifies the backup location.

dwMDVersion: An integer value specifying the version number of the backup to be deleted from the backup location, or the following constant.

Value	Meaning
MD_BACKUP_HIGHEST_VERSION 0xFFFFFFFE	Delete the highest existing backup version in the specified backup location.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the

method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 31.

When processing this call the server MUST do the following:

- Check the *pszMDBackupLocation* parameter backup location. It SHOULD point to the existing place in the configuration store or file system. The exact meaning of this is implementation dependent. Return an error if the location does not exist.
- Check the *dwMDVersion* parameter. If this parameter is equal to MD_BACKUP_HIGHEST_VERSION version number, find and delete the very last backup. Otherwise, find and delete the backup with the requested version. If a backup does not exist, return an error.

3.1.4.16 ChangePermissions (Opnum 19)

The **ChangePermissions** method changes permissions on an open handle.

```
HRESULT ChangePermissions(
  [in] METADATA_HANDLE hMDHandle,
  [in] DWORD dwMDTimeOut,
  [in] DWORD dwMDAccessRequested
);
```

hMDHandle: An unsigned 32-bit integer value containing the handle to change the permissions for, as returned by the **OpenKey** method.

dwMDTimeOut: An integer value specifying the time, in milliseconds, for the method to wait on a successful permission change operation.

dwMDAccessRequested: A set of bit flags specifying the requested permissions for the handle. This parameter must be set to at least one of the following values.

Value	Meaning
METADATA_PERMISSION_READ 0x00000001	Open the node for reading.
METADATA_PERMISSION_WRITE 0x00000002	Open the node for writing.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 19.

If the handle was opened with write permission and is changed to read-only, this method will cause any registered callback methods to be called.

3.1.4.17 GetDataPaths (Opnum 16)

The **GetDataPaths** method returns the paths of all nodes in the subtree relative to a specified starting node that contains the supplied identifier.

```
HRESULT GetDataPaths(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] DWORD dwMDIdentifier,
  [in] DWORD dwMDDataType,
  [in] DWORD dwMDBufferSize,
  [out, size_is(dwMDBufferSize)]
    WCHAR* pszBuffer,
  [out] DWORD* pdwMDRequiredBufferSize
);
```

hMDHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node to be opened, relative to the *hMDHandle* parameter.

dwMDIdentifier: An integer value identifying the data associated with the node.

dwMDDataType: An integer value specifying a data type. If this parameter is not set to ALL_METADATA, the data item will be returned only if its data type matches the specified type.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.
BINARY_METADATA 0x00000003	Specifies binary data in any form.
DWORD_METADATA 0x00000001	Specifies all **DWORD** (unsigned 32-bit integer) data.
EXPANDSZ_METADATA 0x00000004	Specifies all data consisting of a string that includes the NULL-terminating character, which contains unexpanded environment variables.
MULTISZ_METADATA 0x00000005	Specifies all data represented as an array of strings, where each string contains two occurrences of the NULL-terminating character.

Value	Meaning
STRING_METADATA 0x00000002	Specifies all data consisting of an ASCII string that includes the NULL-terminating character.

dwMDBufferSize: An integer value specifying the size, in bytes, of *pbMDBuffer*.

pszBuffer: A pointer to a buffer that contains the retrieved data. If the method call is successful, the buffer will contain an array of **METADATA_GETALL_RECORD** structures.

pdwMDRequiredBufferSize: A pointer to an integer value that contains the buffer length required, in bytes.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x80070008 ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
0x8007007A ERROR_INSUFFICIENT_BUFFER	The data area passed to a system call is too small.

The opnum field value for this method is 16.

When processing this call the server MUST do the following:

1. Check the handle parameter that is passed as the parent handle. This handle could be either the root handle or a handle returned from the previous **OpenKey** call. If the handle is invalid, return an error.
2. Check if the parameter *pszBuffer* is NULL; if it is NULL, then perform the following procedure to calculate the required buffer size and set *pdwMDRequiredBufferSize* to that calculated value. Otherwise, perform the procedure and return the data.
3. Check that the relative path points to a valid node. Otherwise, return an error.
4. On the destination node, find data based on the data ID and the data type. If the data type is set to anything but ALL_METADATA, check that the found data type is the same as the requested parameter. Otherwise, return an error. If the data cannot be found, return an error.
5. For all nodes below the destination node, repeat the same procedure: Find the data by data ID and data type. If the data is available, check its inheritance flag. If the data is inherited, skip to the next node. If the data is not inherited, append the node path to the buffer, if buffer is not NULL. Add the node path length to the required buffer size. Each new path is appended to the buffer in "multi-string" format: Each string is separated by the NULL character, and an extra NULL character is added at the end of buffer after the last string.

3.1.4.18 GetDataSetNumber (Opnum 23)

The **GetDataSetNumber** method returns all the dataset numbers associated with a node in the metabase. A dataset number is a unique number identifying the data items at that node, including inherited data items. Nodes with the same dataset number have identical data.

```
HRESULT GetDataSetNumber(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [out] DWORD* pdwMDDataSetNumber
);
```

hMDHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node to have its dataset number retrieved, relative to the path of *hMDHandle* parameter.

pdwMDDataSetNumber: A pointer to an integer value that returns the number associated with this dataset. You can use this value to identify datasets common to multiple nodes.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000003 ERROR_PATH_NOT_FOUND	The system cannot find the path specified.
0x80000003 E_INVALIDARG	One or more arguments are invalid.
0x80004005 E_FAIL	An unspecified error occurred.
0x800CC800 MD_ERROR_NOT_INITIALIZED	Metadata has not been initialized.

The opnum field value for this method is 23.

3.1.4.19 GetHandleInfo (Opnum 21)

The **GetHandleInfo** method returns information associated with the specified metabase handle.

```
HRESULT GetHandleInfo(
  [in] METADATA_HANDLE hMDHandle,
  [out] METADATA_HANDLE_INFO* pmdhiInfo
```

```
);
```

hMDHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pmdhiInfo: A pointer to a **METADATA_HANDLE_INFO** structure containing information about the specified handle.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000006 ERROR_INVALID_HANDLE	The handle is invalid.
0x80000003 E_INVALIDARG	One or more arguments are invalid.

The opnum field value for this method is 21.

The **dwMDSystemChangeNumber** member of the **METADATA_HANDLE_INFO** structure pointed to by *pmdhiInfo* parameter will correspond to the system change number generated at the time the handle was created. It will not change if write operations are performed by using this handle, or any other handle. This number can be compared with the value returned by the **GetSystemChangeNumber** method to see if any write operations have been performed since the handle was opened.

3.1.4.20 GetLastChangeTime (Opnum 25)

The **GetLastChangeTime** method returns the last change time associated with a node in the metabase.

```
HRESULT GetLastChangeTime(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [out] PFILETIME pftMDLastChangeTime,
  [in] BOOL bLocalTime
);
```

hMDHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string containing the path of the node to be set, relative to the path of *hMDHandle* parameter.

pftMDLastChangeTime: A pointer to a **FILETIME** structure that returns the last change time for the node.

bLocalTime: A Boolean value indicating whether the time value returned in *pftMDLastChangeTime* parameter is specified as local time (TRUE) or Coordinated Universal Time (UTC) time (FALSE).

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x80000003 E_INVALIDARG	One or more arguments are invalid.
0x8000FFFF E_UNEXPECTED	A catastrophic failure occurred.

The opnum field value for this method is 25.

3.1.4.21 GetSystemChangeNumber (Opnum 22)

The **GetSystemChangeNumber** method returns the number of changes made to data since the metabase was created.

```
HRESULT GetSystemChangeNumber(
  [out] DWORD* pdwSystemChangeNumber
);
```

pdwSystemChangeNumber: A pointer to an unsigned 32-bit integer value containing the system change number. This number is increased each time the metabase is updated.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x80000003 E_INVALIDARG	One or more arguments are invalid.

The opnum field value for this method is 22.

3.1.4.22 R_GetAllData (Opnum 13)

The **R_GetAllData** method returns all data associated with a node in the metabase, including all values that the node inherits.

```
HRESULT R_GetAllData(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] DWORD dwMDAttributes,
  [in] DWORD dwMDUserType,
  [in] DWORD dwMDDataType,
  [out] DWORD* pdwMDNumDataEntries,
  [out] DWORD* pdwMDDataSetNumber,
  [in] DWORD dwMDBufferSize,
  [out] DWORD* pdwMDRequiredBufferSize,
  [out] IIS_CRYPTO_BLOB** ppDataBlob
);
```

hMDHandle: An unsigned 32-bit integer value containing the handle to a node in the metabase. The handle can be the metadata master root handle (0x00000000) or a handle, with read permission, returned by the **OpenKey** method.

pszMDPath: A pointer to a Unicode string that contains the path of the node with which the data to be returned is associated, relative to the path of *hMDHandle* parameter.

dwMDAttributes: Flags used to specify the data, as listed in the **METADATA_RECORD** structure.

dwMDUserType: An integer value specifying the data to return based on user type.

Value	Meaning
ALL_METADATA 0x00000000	Returns all data, regardless of user type.
ASP_MD_UT_APP 0x00000065	Returns data specific to ASP application configuration.
IIS_MD_UT_FILE 0x00000002	Returns data specific to a file, such as access permissions or logon methods.
IIS_MD_UT_SERVER 0x00000001	Returns data specific to the server, such as ports in use and IP addresses.
IIS_MD_UT_WAM 0x00000064	Returns data specific to Web application management.

dwMDDataType: An integer value specifying a data type. If this parameter is not set to ALL_METADATA, the data item will be returned only if its data type matches the specified type.

Value	Meaning
ALL_METADATA 0x00000000	Specifies all data, regardless of type.

Value	Meaning
BINARY_METADATA 0x00000003	Specifies binary data in any form.
DWORD_METADATA 0x00000001	Specifies all **DWORD** (unsigned 32-bit integer) data.
EXPANDSZ_METADATA 0x00000004	Specifies all data that consists of a null-terminated string containing environment variables that are not expanded.
MULTISZ_METADATA 0x00000005	Specifies all data represented as an array of null-terminated strings, terminated by two null characters.
STRING_METADATA 0x00000002	Specifies all data consisting of a null-terminated ASCII string.

pdwMDNumDataEntries: A pointer to an integer value that contains the number of entries copied to *pbBuffer* parameter.

pdwMDDataSetNumber: A pointer to an integer value used to identify this dataset.

dwMDBufferSize: An integer value specifying the size, in bytes, of *pbMDBuffer* parameter.

pdwMDRequiredBufferSize: A pointer to a buffer that contains the retrieved data. If the method call is successful, the buffer will contain an array of **METADATA_GETALL_RECORD** structures.

ppDataBlob: An **IIS_CRYPTO_BLOB** structure containing the requested values as encrypted opaque data.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Note Invalid *dwMDUserType* or *dwMDDataType* parameters result in a return status of 0x80070057 = E_INVALIDARG.

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 13.

On non-Intel platforms, the **DWORD** data is aligned; however, this may not be true on remote clients.

The server MUST check whether at least one of the **METADATA_RECORD** entries contains sensitive data. The METADATA_SECURE secure flag in the *dwMDAttributes* member of the **METADATA_RECORD** structure for all entries will be set.

If at least one matching entry with the METADATA_SECURE flag set is found:

- Encrypt the data value based on the procedure described in section 3.1.4.1.2. The encrypted data BLOB will be stored in the **IIS_CRYPTO_BLOB** message format with the **BlobSignature** field set to the **ENCRYPTED_DATA_BLOB_SIGNATURE** signature.

If no **METADATA_RECORD** entry with the METADATA_SECURE flag was found:

- Build the **IIS_CRYPTO_BLOB** message with the **BlobSignature** field set to CLEARTEXT_DATA_BLOB_SIGNATURE. Store the cleartext data in the **BlobData** field. Set the **BlobDataLength** field to match the length of the **BlobData** field.

3.1.4.23 R_GetData (Opnum 10)

The **R_GetData** method returns a data entry from a particular node in the metabase.

```
HRESULT R_GetData(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in, out] METADATA_RECORD* pmdrMDData,
  [out] DWORD* pdwMDRequiredDataLen,
  [out] IIS_CRYPTO_BLOB** ppDataBlob
);
```

hMDHandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node containing the data, relative to the path of *hMDHandle* parameter.

pmdrMDData: A pointer to a **METADATA_RECORD** structure that contains the requested data.

pdwMDRequiredDataLen: An integer value specifying the data length, in bytes, of the required buffer size.

ppDataBlob: An **IIS_CRYPTO_BLOB** structure containing the requested values as encrypted opaque data.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 10.

The *pbMDData* member of **METADATA_RECORD** is not used to transfer the actual data value with **R_GetData**. The *pbMDData* parameter SHOULD be set to NULL. The **IIS_CRYPTO_BLOB** structure is used to transfer the actual data message received by the server and can be encrypted when server sends data marked as secure.

The following set of steps MUST be performed by server to encrypt or encode a data value and build a **IIS_CRYPTO_BLOB** structure to be sent by server.

1. Check if the *dwMDAttributes* member of the **METADATA_RECORD** structure has the METADATA_SECURE secure flag set.

2. If the METADATA_SECURE secure flag is set, then:

   - Encrypt the data value based on the procedure described in section 3.1.4.1.2. The encrypted data BLOB will be stored in the **IIS_CRYPTO_BLOB** message format with the **BlobSignature** field set to the ENCRYPTED_DATA_BLOB_SIGNATURE signature.

3. If the METADATA_SECURE secure flag is not set, then:

   - Build the **IIS_CRYPTO_BLOB** message with the **BlobSignature** field set to CLEARTEXT_DATA_BLOB_SIGNATURE. Store the cleartext data in the **BlobData** field. Set the **BlobDataLength** field to match the length of the **BlobData** field.

3.1.4.24 R_GetServerGuid (Opnum 33)

The **R_GetServerGuid** method returns the Globally Unique Identifier (GUID) for the running Internet Information Services (IIS) instance.

```
HRESULT R_GetServerGuid(
  [out] GUID* pServerGuid
);
```

pServerGuid: A **GUID** uniquely identifying the current Internet Information Services (IIS) instance.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 33.

3.1.4.25 R_KeyExchangePhase1 (Opnum 26)

The **R_KeyExchangePhase1** method receives a pair of encrypted client keys and returns server encryption and session keys.

```
HRESULT R_KeyExchangePhase1(
  [unique, in] IIS_CRYPTO_BLOB* pClientKeyExchangeKeyBlob,
  [unique, in] IIS_CRYPTO_BLOB* pClientSignatureKeyBlob,
  [out] IIS_CRYPTO_BLOB** ppServerKeyExchangeKeyBlob,
  [out] IIS_CRYPTO_BLOB** ppServerSignatureKeyBlob,
```

```
  [out] IIS_CRYPTO_BLOB** ppServerSessionKeyBlob
);
```

pClientKeyExchangeKeyBlob: A pointer to an **IIS_CRYPTO_BLOB** structure containing the encrypted client key used to decrypt client data.

pClientSignatureKeyBlob: A pointer to an **IIS_CRYPTO_BLOB** structure containing the encrypted client signature key used for data verification.

ppServerKeyExchangeKeyBlob: A pointer to a set of **IIS_CRYPTO_BLOB** structures containing encrypted server keys used by the client to decrypt server data.

ppServerSignatureKeyBlob: A pointer to a set of **IIS_CRYPTO_BLOB** structures containing encrypted server signature keys used for data verification.

ppServerSessionKeyBlob: A pointer to a set of **IIS_CRYPTO_BLOB** structures containing encrypted server session keys.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 26.

When the server receives the **R_KeyExchangePhase1** method, it SHOULD check the state of the secure session. If the session was already negotiated, then the server SHOULD return the E_INVALIDARG error code.

If the session was not negotiated yet, then the server MUST perform the following steps:

1. Store the client's Key Exchange Public Key that was received in the message.
2. Store the client's Signature Public Key that was received in the message.
3. Generate or locate the server's Key Exchange Private Key.
4. Generate or locate the server's Signature Private Key.
5. Generate the server's Session Key.
6. Encrypt the server's Session Key using the client's Key Exchange Public Key that was just received.
7. Calculate the signed hash of the Encrypted Server's Session Key, as described in section 3.1.4.1.4. Use the server's Signature Private Key for signing.

8. Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to SESSION_KEY_BLOB_SIGNATURE to store the server's encrypted session key and signed hash as calculated in the previous steps.

9. Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to PUBLIC_KEY_BLOB_SIGNATURE to store the server's Key Exchange Public Key.

10. Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to PUBLIC_KEY_BLOB_SIGNATURE to store the server's Signature Public Key.

11. Send an **IIS_CRYPTO_BLOB** structure that was built in the previous three steps to the client in response to the **R_KeyExchangePhase1** method.

3.1.4.26 R_KeyExchangePhase2 (Opnum 27)

The **R_KeyExchangePhase2** method receives the encrypted client session and hash keys in response to the **R_KeyExchangePhase1** method and returns the encrypted server hash keys.

```
HRESULT R_KeyExchangePhase2(
  [unique, in] IIS_CRYPTO_BLOB* pClientSessionKeyBlob,
  [unique, in] IIS_CRYPTO_BLOB* pClientHashBlob,
  [out] IIS_CRYPTO_BLOB** ppServerHashBlob
);
```

pClientSessionKeyBlob: A pointer to an **IIS_CRYPTO_BLOB** structure containing the encrypted client session key.

pClientHashBlob: A pointer to an **IIS_CRYPTO_BLOB** structure containing the encrypted client hash key.

ppServerHashBlob: A pointer to a set of **IIS_CRYPTO_BLOB** structures containing encrypted session hash keys.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 27.

When the server receives the **R_KeyExchangePhase2** method, it SHOULD check the state of the secure session. If the session negotiation has not started yet by processing the **R_KeyExchangePhase1** method, then the MD_ERROR_SECURE_CHANNEL_FAILURE error code MUST be returned back to client.

If any of the parameters sent by the client are empty, then the server SHOULD return an E_INVALIDARG error code.

Upon successful validation, the server SHOULD perform the following steps:

1. Decrypt the encrypted client's Session Key using the server's Key Exchange Private Key
2. Store the client's Session Key
3. Compute the hash of the following 3 values (in this order):
    1. Client's Session Key
    2. Server's Session Key
    3. Value of HASH_TEXT_STRING_1, as specified in section 2.2.2.
4. Compare the hash computed in the previous step with the hash received from the client. If they match, then the client has proved that it owns the client's Key Exchange Private Key that matches the client's Key Exchange Public Key. It proved it by being able to decrypt the server's Session Key that was needed for the hash calculation.
5. Compute hash of the following 2 values (in this order):
    1. Client's Session Key
    2. Value of HASH_TEXT_STRING_2, as specified in 2.2.2.
6. Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to HASH_BLOB_SIGNATURE and store the hash calculated in the previous step.
7. Send the **IIS_CRYPTO_BLOB** structure calculated in the previous step to the client.

3.1.4.27 R_SetData (Opnum 9)

The **R_SetData** method sets a data item for a particular node in the metabase.

```
HRESULT R_SetData(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] METADATA_RECORD* pmdrMDData
);
```

hMDHandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with write permissions as returned by the **OpenKey** method. This handle SHOULD not be the master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node that stores the entry, relative to the path of *hMDHandle* parameter.

pmdrMDData: A pointer to a **METADATA_RECORD** structure that contains the data to set.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 1627) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 9.

If data with the specified identifier does not exist, this method creates and inserts a data item into the list of data items of that type. If data with the specified identifier does exist, this method sets the new data value. Duplicate data identifiers are not valid even if the entries are of different user types or data types.

The *pbMDData* and *dwMDDataLen* fields of **METADATA_RECORD** (referenced by the *pmdrMDData* parameter) MUST be interpreted using following steps:

1. Check if the *dwMDAttributes* member of the **METADATA_RECORD** structure has the METADATA_SECURE secure flag set.

2. If the METADATA_SECURE secure flag is set, then the *pbMDData* member of **METADATA_RECORD** structure points to the encrypted data BLOB and the *dwMDDataLen* field is set to the size of the encrypted data. The encrypted data BLOB is stored in the **IIS_CRYPTO_BLOB** message format with the **BlobSignature** field set to ENCRYPTED_DATA_BLOB_SIGNATURE.

   - Decrypt the data pointed to by the *pbMDData* field based on the procedure described in section 3.1.4.1.3.

3. If the METADATA_SECURE secure flag is not set, then the data referenced by the *pbMDData* member of **METADATA_RECORD** is the cleartext data and *dwMDDataLen* field is its length.

3.1.4.28 RenameKey (Opnum 8)

The **RenameKey** method renames a node in the metabase.

```
HRESULT RenameKey(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [unique, in, string] LPCWSTR pszMDNewName
);
```

hMDHandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with write permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string that contains the path of the node to be renamed, relative to the path of *hMDHandle* parameter.

pszMDNewName: A pointer to a string that contains the new name for the node.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x80000002 E_OUTOFMEMORY	There was not enough memory to complete the method call.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function, CoInitialize, has not been called. For more information, see [MSDN-CoInitialize].

The opnum field value for this method is 8.

When processing this call, the server MUST do the following:

- Check the handle parameter. This handle SHOULD be the root handle or a handle returned from the **OpenKey** call. If the handle is invalid, return an error.
- Check if the parent handle was opened for write access; otherwise, return an error.
- Check if the relative path has the right syntax and length. If not, return an error.
- Check if the caller has write access to the destination node. If not, return an error.
- Check if the caller has write access to the node with the resulting path. If not, return an error.
- Check if there is an existing node with the resulting path. Return an error if true.
- Without modifying the data, replace the target node name with the new name.

3.1.4.29 Restore (Opnum 29)

The **Restore** method restores the metabase from a backup.

```
HRESULT Restore(
  [unique, in, string] LPCWSTR pszMDBackupLocation,
  [in] DWORD dwMDVersion,
  [in] DWORD dwMDFlags
);
```

pszMDBackupLocation: A string of up to 100 Unicode characters that identifies the backup location containing the backup to be restored.

dwMDVersion: An integer value specifying the version number of the backup to be restored from the backup location, or the following constant.

Value	Meaning
MD_BACKUP_HIGHEST_VERSION 0xFFFFFFFE	Restore from the highest existing backup version in the specified backup location.

dwMDFlags: This parameter is reserved and SHOULD always be set to zero.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value

contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000057 ERROR_INVALID_PARAMETER	The parameter is incorrect.
0x800CC801 MD_ERROR_DATA_NOT_FOUND	The specified metadata was not found.

The opnum field value for this method is 29.

When processing this call the server MUST do the following:

- The server restores from a backup that is identified by the *pszMDBackupLocation* parameter and the version number.
- If the *pszMDBackupLocation* parameter is an empty string, the server MUST use the default backup location as defined by the server implementation.
- If the backup location contains illegal characters as defined by the server-specific implementation, the server MUST return the HRESULT derived from ERROR_INVALID_NAME error code.
- If the backup location does not exist, the server MUST return the E_INVALIDARG error code.
- If the value of the *dwMDVersion* parameter is greater than MD_BACKUP_MAX_VERSION (9999) and not equal to MD_BACKUP_HIGHEST_VERSION, the server MUST return the E_INVALIDARG error code.
- If the *dwMDVersion* parameter equals MD_BACKUP_HIGHEST_VERSION, the server MUST restore the metadata to the metadata stored in the highest version at this location.

3.1.4.30 SaveData (Opnum 20)

The **SaveData** method explicitly flushes the metabase data resident in memory to configuration storage.

```
HRESULT SaveData();
```

This method has no parameters.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x800401F0 CO_E_NOTINITIALIZED	The Component Object Model (COM) function, CoInitialize, has not been called. For more information, see [MSDN-CoInitialize].

The opnum field value for this method is 20.

All data in the metabase is saved, including data written by other applications.

A handle MUST NOT be open with write permission when this method is called, or the method will fail. The process waits for a few seconds for handles to close, so other processes with open write handles SHOULD NOT normally interfere with the save operation.

3.1.4.31 SetLastChangeTime (Opnum 24)

The **SetLastChangeTime** method sets the last change time associated with a node in the metabase.

```
HRESULT SetLastChangeTime(
  [in] METADATA_HANDLE hMDHandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] PFILETIME pftMDLastChangeTime,
  [in] BOOL bLocalTime
);
```

hMDHandle: An unsigned 32-bit integer value containing a handle to a node in the metabase with write permissions as returned by the **OpenKey** method, or the metabase master root handle (0x00000000).

pszMDPath: A pointer to a Unicode string containing the path of the node to be set, relative to the path of *hMDHandle*.

pftMDLastChangeTime: A pointer to a **FILETIME** structure that contains the last change time to set for the node.

bLocalTime: A Boolean value indicating whether the time value specified in *pftMDLastChangeTime* parameter is local time (TRUE) or Coordinated Universal Time (UTC) time (FALSE).

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000003	The system cannot find the path specified.

Return value/code	Description
ERROR_PATH_NOT_FOUND	
0x80000003 E_INVALIDARG	One or more arguments are invalid.
0x800CC800 MD_ERROR_NOT_INITIALIZED	Metadata has not been initialized.

The opnum field value for this method is 24.

3.1.4.32 UnmarshalInterface (Opnum 32)

The **UnmarshalInterface** method returns the pointer to the **IMSAdminBaseW** interface.

```
HRESULT UnmarshalInterface(
  [out] IMSAdminBaseW** piadmbwInterface
);
```

piadmbwInterface: The address of the pointer that contains the pointer to **IMSAdminBaseW**.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

The opnum field value for this method is 32.

3.1.5 Timer Events

No protocol timer events are required on the server other than the timers that are required in the underlying RPC protocol.

3.1.6 Other Local Events

No local events are maintained on the server other than the events that are maintained in the underlying RPC protocol.

3.2 IMSAdminBaseW Client Role Details

3.2.1 Abstract Data Model

The client SHOULD use the abstract data model defined by the server; see section 3.1.1.

3.2.1.1 Secure Session Context

When the client expects to exchange sensitive data marked with the METADATA_SECURE secure flag, it will negotiate a secure session. As part of the secure session negotiation, both client and server will build the secure session context.

Each client MUST maintain the following information related to the secure session:

- The client's Key Exchange Private and Public Key.
- The client's Signature Private and Public Key.
- The server's Key Exchange Public Key.
- The server's Signature Public Key.
- The server's Session Key.
- The client's Session Key.

3.2.2 Timers

No protocol timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3.2.3.2.1.

3.2.3 Initialization

The client MUST perform initialization according to the following rules when calling an RPC method:

- Either create an RPC binding handle to the server or use an RPC context handle. Details concerning binding handles are as specified in [C706] section 2.3.
- Use context handles across multiple calls to server for methods taking METADATA_HANDLE as a parameter.
- A context handle SHOULD be reused in multiple invocations when getting or setting information to remote server configuration.
- When creating the RPC binding handle, the client MUST specify an **ImpersonationLevel** of 2 (Impersonation), as specified in [MS-DCOM].

3.2.4 Message Processing Events and Sequencing Rules

3.2.4.1 Secure Session Negotiation Client Role

The client MUST negotiate a secure session when sensitive data is to be transferred; for more information, see 3.1.4.1.1.

The client performs the secure session negotiation by processing the **R_KeyExchangePhase1** and **R_KeyExchangePhase2** calls, as described in sections 3.2.4.2 and 3.2.4.3.

As a result of secure session negotiation, the client will receive the server's Session Key and Signature Public Key.

3.2.4.2 R_KeyExchangePhase1(Opnum 26)

The client MUST perform the following steps to process **R_KeyExchangePhase1**:

- Generate or look up the client's Key Exchange Key.
- Generate or look up the client's Signature Key.
- Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to PUBLIC_KEY_BLOB_SIGNATURE to store the client's Key Exchange Public Key.
- Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to PUBLIC_KEY_BLOB_SIGNATURE to store the client's Signature Public Key.
- Send the **IIS_CRYPTO_BLOBs** built in the previous two steps to the server by using **R_KeyExchangePhase1** method.
- Wait for the response from the server.
- Retrieve the server's Key Exchange Public Key from the **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to PUBLIC_KEY_BLOB_SIGNATURE.
- Retrieve the server's Signature Public Key from **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to PUBLIC_KEY_BLOB_SIGNATURE.
- Retrieve the server's Session Key from the **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to SESSION_KEY_BLOB_SIGNATURE.
- Decrypt the server's Session Key by using the client's Key Exchange Private Key.
- In the case of success, the client MUST proceed with the **R_KeyExchangePhase2** method.

3.2.4.3 R_KeyExchangePhase2 (Opnum 27)

The client MUST call R_KeyExchangePhase2 after successful processing of R_KeyExchangePhase1 to complete the security session negotiation.

The client MUST perform the following steps:

- Generate and store the client's Session Key.
- Encrypt the client's Session Key with the server's Key Exchange Public Key.
- Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to SESSION_KEY_BLOB_SIGNATURE to store the encrypted client's Session Key.
- Compute the hash of the following three entities in this order:
  - The client's Session Key.
  - The server's Session Key.
  - The text value of HASH_TEXT_STRING_1, as specified in section 2.2.2.
- Build an **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to SESSION_BLOB_SIGNATURE to store the hash computed in the previous step.
- Send the **IIS_CRYPTO_BLOBs** built in the previous steps to the server.
- Wait for the response from the server.

- Retrieve the server's Hash **IIS_CRYPTO_BLOB** with the **BlobSignature** field set to HASH_BLOB_SIGNATURE sent by the server.
- Build the hash for validation purposes.
- Compute the hash of the client's Session Key.
- Compute the hash of the text HASH_TEXT_STRING_2, as specified in section 2.2.2.
- Compare hashes from the previous two steps. If they match, then the server owns the server's Key Exchange Private Key and was able to decrypt the client's Session Key.
- Secure session negotiation is now complete. The client and server can now use secure session to encrypt/decrypt data of a sensitive nature marked by the METADATA_SECURE secure flag with calls to **R_GetData**, **R_EnumData**, **R_GetAllData**, and **R_SetData** methods.

3.2.4.4 R_SetData (Opnum 9)

The data value referenced by the **pbMDData** field of the **METADATA_RECORD** MUST be encrypted if the METADATA_SECURE attribute is set.

- Check if the **dwMDAttributes** member of the **METADATA_RECORD** structure has a METADATA_SECURE flag set.
- If the METADATA_SECURE secure flag is set:
  - Negotiate secure session (see section 3.1.4.1.1) if it was not negotiated yet.
  - Encrypt the data value based on the procedure described in section 3.1.4.1.2. The encrypted data blob will be stored in the **IIS_CRYPTO_BLOB** message format with the **BlobSignature** field set to ENCRYPTED_DATA_BLOB_SIGNATURE.
  - Set the **pbMDData** and **dwMDDataLen** fields in the **METADATA_RECORD** message (referenced by **pmdrMDData**). The **pbMDData** field MUST be updated to point to the **IIS_CRYPTO_BLOB** message built in the previous step. The **dwMDDataLen** field MUST be set to the total length in bytes of the **IIS_CRYPTO_BLOB** message built in the previous step.
- If the METADATA_SECURE flag is not set, then the cleartext data value will be referenced by **pbMDData** field of the **METADATA_RECORD** message and the **dwMDDataLen** field will be set to the length of that cleartext data.

3.2.4.5 R_GetData (Opnum 10)

The secure session MUST be negotiated by the client prior to calling R_GetData (see 3.1.4.1.1).

The **pbMDData** field of the **METADATA_RECORD** structure is not used for the R_GetData call.

The **IIS_CRYPTO_BLOB** received by the client upon successful completion of the call to R_GetData contains encrypted or encoded data.

The following set of steps MUST be performed by the client to decrypt or decode **IIS_CRYPTO_BLOB** data received from the server:

- If the **BlobSignature** in the **IIS_CRYPTO_BLOB** message is ENCRYPTED_DATA_BLOB_SIGNATURE, then data inside the message will be decrypted based on the description in section 3.1.4.1.2.

- If the **BlobSignature** in the **IIS_CRYPTO_BLOB** message is CLEARTEXT_DATA_BLOB_SIGNATURE, then **BlobData** inside the **IIS_CRYPTO_BLOB** will be interpreted as a CLEARTEXT_DATA_BLOB message. The **CleartextData** field represents the actual clear text data.

3.2.4.6 R_EnumData (Opnum 12)

The session negotiation requirement, **IIS_CRYPTO_BLOB** handling, and **METADATA_RECORD** handling is identical to that used by R_GetData.

3.2.4.7 R_GetAllData (Opnum 13)

The secure session MUST be negotiated by the client prior to calling R_GetAllData (see 3.1.4.1.1).

The **IIS_CRYPTO_BLOB** received by the client upon successful completion of the R_GetAllData contains encrypted or encoded data.

The following set of steps MUST be performed by the client to decrypt or decode **IIS_CRYPTO_BLOB** data received from the server.

- If **BlobSignature** in the **IIS_CRYPTO_BLOB** message is ENCRYPTED_DATA_BLOB_SIGNATURE, then data inside the message MUST be decrypted based on the description in section 3.1.4.1.2.
- If **BlobSignature** in the **IIS_CRYPTO_BLOB** message is CLEARTEXT_DATA_BLOB_SIGNATURE, then **BlobData** inside the **IIS_CRYPTO_BLOB** will be interpreted as CLEARTEXT_DATA_BLOB message. The **ClearTextData** field represents the actual clear text data.

The clear text data retrieved in previous steps follows the **METADATA_GETALL_RECORD** format.

3.2.5 Timer Events

No protocol timer events are required on the client beyond the timers required in the underlying RPC protocol.

3.2.6 Other Local Events

A client's invocation of each method is typically the result of local application activity. The local application on the client computer specifies values for all input parameters. No other higher-layer triggered events are processed. The values specified for input parameters are described in section 2.

No additional local events are used on the client beyond the events maintained in the underlying RPC protocol.

3.3 IMSAdminBase2W Server Role Details

3.3.1 Abstract Data Model

This interface uses the same data model as the **IMSAdminBaseW** interface.

3.3.2 Timers

No protocol timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3.2.3.2.1.

3.3.3 Initialization

Initialization is specified in section 3.1.3.

3.3.4 Message Processing Events and Sequencing Rules

This DCOM interface inherits the IUnknown interface. The method opnum field values start with 3; opnum values 0 through 2 represent the IUnknown_QueryInterface, IUnknown_AddRef, and IUnknown_Release methods, respectively, as specified in [MS-DCOM].

Methods with opnum field values 3 through 33 are defined in section 3.1.4, and field value 40 is defined in section 3.5.4.

The **IMSAdminBase2W** Remote Procedure Call (RPC) interface extends the **IMSAdminBaseW** interface, adding functionality for metabase importing and exporting, history management, and secure data encryption on backup. The **IMSAdminBase2W** protocol does not maintain client state information.

An RPC sequence is a client/server session that includes a security context phase and requests to call remote procedures. For connection-oriented RPC, the session also includes a binding phase. The RPC client supplies the necessary security information and for a connection-oriented RPC, it also supplies binding information such as interface name and server **endpoint**. The sequence of subsequent RPC calls in the session is implementation-specific.

Methods in RPC Opnum Order

Method	Description
BackupWithPasswd	Backs up the metabase to a specified location, using a supplied password to encrypt all secure data. Opnum: 34
RestoreWithPasswd	Restores the metabase from a backup, using a supplied password to decrypt the secure data. Opnum: 35
Export	Exports the metabase from a supplied location to a specific file name. Opnum: 36
Import	Imports a previously exported metabase into an existing one. Opnum: 37
RestoreHistory	Restores a metabase history entry for a specific history version. Opnum: 38
EnumHistory	Returns an enumerated history entry with a supplied index. Opnum: 39

When a remote call is made, the UUID and version number of the interface are specified in the **abstract_syntax** and **abstract_syntax_vers** fields of the incoming RPC_BIND packet, as specified in [MS-RPCE].

3.3.4.1 BackupWithPasswd (Opnum 34)

The **BackupWithPasswd** method backs up the metabase to a specified location, using a supplied password to encrypt all secure data.

```
HRESULT BackupWithPasswd(
  [unique, in, string] LPCWSTR pszMDBackupLocation,
  [in] DWORD dwMDVersion,
  [in] DWORD dwMDFlags,
  [unique, in, string] LPCWSTR pszPasswd
);
```

pszMDBackupLocation: The name of the backup that is being created. An empty string SHOULD be interpreted as a request to use a default name for the backup. The *pszMDBackupLocation* parameter MUST be equal to or less than MD_BACKUP_MAX_LEN (100) **WCHAR** characters in length.

dwMDVersion: An integer value specifying either the specific version number to be used for the backup or one of the following flag values. If the version number is an explicit version number it SHOULD be less than MD_BACKUP_MAX_VERSION (9999).

Value	Meaning
MD_BACKUP_HIGHEST_VERSION 0xFFFFFFFE	Use the highest existing backup version for the backup name specified.
MD_BACKUP_NEXT_VERSION 0xFFFFFFFF	Use the highest existing backup version number plus one for the backup name specified.

dwMDFlags: An integer value containing the bit flags to alter backup functionality. The flags can be zero or one or more of the following values.

Value	Meaning
MD_BACKUP_FORCE_BACKUP 0x00000004	Force the backup even if the **SaveData** operation specified by MD_BACKUP_SAVE_FIRST fails. This flag SHOULD only be specified if MD_BACKUP_SAVE_FIRST is specified.
MD_BACKUP_OVERWRITE 0x00000001	Back up even if a backup of the same name and version exists in the specified backup location, overwriting it if necessary.
MD_BACKUP_SAVE_FIRST 0x00000002	Perform a **SaveData** operation before the backup.

pszPasswd: A password string used to encrypt the secure properties in the metabase backup. If a password is not supplied, this method functions exactly the same as **Backup**.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000003 ERROR_PATH_NOT_FOUND	The system cannot find the path specified.
0x00000057 ERROR_INVALID_PARAMETER	The parameter is incorrect.
0x00000094 ERROR_PATH_BUSY	The path specified cannot be used at this time.
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.
0x80070008 HRESULT derived from ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
0x8007007B HRESULT derived from ERROR_INVALID_NAME	The name of the backup is invalid.
0x80070005 HRESULT derived from ERROR_ACCESS_DENIED	Access is denied.
0x800CC800 MD_ERROR_NOT_INITIALIZED	Metadata has not been initialized.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol [MS-RPCE].

The opnum field value for this method is 34.

When processing this call the server MUST do the following:

- Check the parameter backup location. It SHOULD point to the existing place in the configuration store or file system. The exact meaning of this is implementation-dependent. Return an error if the location does not exist.
- Check the version parameter. If it is greater than the maximum allowed version number and is not MD_BACKUP_HIGHEST_VERSION or MD_BACKUP_NEXT_VERSION, the server returns the E_INVALIDARG error code.
- If the parameter flags have the MD_BACKUP_SAVE_FIRST bit set, flush the in-memory configuration data first. If this operation fails, check the MD_BACKUP_FORCE_BACKUP bit. If this bit is reset, return an error, otherwise continue the operation.
- Check the MD_BACKUP_OVERWRITE bit. If it is reset, check if a backup with the target version exists. If it is true, return an error, otherwise overwrite the existing backup.
- If the backup location is an empty string, the server backs up the configuration to the default backup location.
- The server saves the persisted data using the backup location and version number as a key so that the data can be restored later.

- Any encrypted data MUST be stored encrypted with the password the client has provided. If no password is provided, the function behaves exactly as the **Backup** method.

3.3.4.2 EnumHistory (Opnum 39)

The **EnumHistory** method returns an enumerated history entry with a supplied index.

```
HRESULT EnumHistory(
  [in, out, size_is(ADMINDATA_MAX_NAME_LEN)]
    LPWSTR pszMDHistoryLocation,
  [out] DWORD* pwdMDMajorVersion,
  [out] DWORD* pwdMDMinorVersion,
  [out] PFILETIME pftdMDHistoryTime,
  [in] DWORD dwMDEnumIndex
);
```

pszMDHistoryLocation: A pointer to a Unicode string which on input contains the path to the history files being enumerated. If this is an empty string, the default path SHOULD be used. If an empty string had been passed in, the default history path will be returned. The underlying buffer SHOULD be 100 characters regardless of whether there are 100 characters in the path or not, because of the possibility of an altered string being returned. <3>

pwdMDMajorVersion: A pointer to an integer value containing the pre-decimal version number for the current enumerated history entry.

pwdMDMinorVersion: A pointer to an integer value containing the post-decimal version number for the current enumerated history entry.

pftdMDHistoryTime: A pointer to a **FILETIME** structure containing the time stamp for the current enumerated history entry.

dwMDEnumIndex: An integer value containing the current index of the history entry to be enumerated. This value SHOULD start at zero on the first call and SHOULD be increased by one on subsequent calls until the last entry in the history is reached. This indexing is controlled by the client, so the client is responsible for selecting the next history file to be enumerated.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000002 HRESULT derived from ERROR_PATH_NOT_FOUND	The system cannot find the file specified.
0x00000008 HRESULT derived from	Not enough storage is available to process this command.

Return value/code	Description
ERROR_NOT_ENOUGH_MEMORY	
0x00000012 HRESULT derived from ERROR_NO_MORE_ITEMS	There are no more history versions.
0x0000007A HRESULT derived from ERROR_INSUFFICIENT_BUFFER	The data area passed to a system call is too small. In this case the location string does not have enough space to return the path to the history location.
0x80070005 HRESULT derived from ERROR_ACCESS_DENIED	Access is denied.
0x80000003 E_INVALIDARG	One or more arguments are invalid.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol [MS-RPCE].

The opnum field value for this method is 39.

When processing this call the server MUST do the following:

- If the *pszMDHistoryLocation* is longer than 100 **WCHARs**, an E_INVALIDARG error code will be returned.
- If the *pszMDHistoryLocation* is an empty string, the default history directory will be used and this value will be written to the *pszMDHistoryLocation* buffer. Therefore, the history location buffer needs to be large enough to hold this string. Thus it is expected that the client be passing a buffer of at least 100 **WCHARs** even in the case where an empty string is passed.
- The server will find the history entry that corresponds to the location passed in and the index number. For instance, it will look for the 3 file in the directory if 4 is passed in for the index (zero-based).
- Once the history entry is found, the server will return the version number of the history entry in the two version parameters. The server also will return the file time stamp information in the *pftdMDHistoryTime* parameter.
- If the index is past the last file in the history location, the server MUST return an ERROR_NO_MORE_ITEMS error code to indicate that there are no more items to the client.

3.3.4.3 Export (Opnum 36)

The **Export** method exports the metabase from a supplied location to a specific file name.

```
HRESULT Export(
  [unique, in, string] LPCWSTR pszPasswd,
  [unique, in, string] LPCWSTR pszFileName,
  [unique, in, string] LPCWSTR pszSourcePath,
  [in] DWORD dwMDFlags
);
```

pszPasswd: A pointer to a Unicode string containing the password that will be used to encrypt the secure properties of the metabase being exported.

pszFileName: A pointer to a Unicode string containing the name of the file, including the directory path, to which the data will be exported. The path SHOULD exist and the path SHOULD be local to the server.

pszSourcePath: A pointer to a Unicode string containing the path to the source metabase node.<4>

dwMDFlags: A set of bit flags specifying the export operation to be performed. It can be zero or one or more of the following values.

Value	Meaning
MD_EXPORT_INHERITED 0x00000001	Settings inherited from the parent nodes will be included in the export.
MD_EXPORT_NODE_ONLY 0x00000002	Child nodes will not be exported.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000008 ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
0x00000057 ERROR_INVALID_PARAMETER	The parameter is incorrect.
0x000003EC ERROR_INVALID_FLAGS	Invalid flags were passed.
0x80000003 E_INVALIDARG	One or more arguments are invalid.
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.
0x800CC800 MD_ERROR_NOT_INITIALIZED	Metadata has not been initialized.
0x80070003 HRESULT derived from ERROR_PATH_NOT_FOUND	The system cannot find the path specified.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol [MS-RPCE].

The opnum field value for this method is 36.

When processing this call the server MUST do the following:

- The path passed as the *pszFileName* parameter SHOULD exist prior to the call. If it does not, the server returns HRESULT derived from the ERROR_PATH_NOT_FOUND error code.
- The path passed as the *pszFileName* parameter SHOULD be a local path to the server. If it does not, the server returns HRESULT derived from the ERROR_PATH_NOT_FOUND error code.
- The server MUST validate that the source path provided does map to a node in the data hierarchy.
- If the MD_EXPORT_INHERITED flag is passed, the server MUST include inherited property values in the exported data.
- If the MD_EXPORT_NODE_ONLY flag is passed, the server MUST include only the specified node and its settings. Child nodes MUST NOT be included.
- Any encrypted data MUST be stored as encrypted with the password that was provided by the client.

3.3.4.4 Import (Opnum 37)

The **Import** method imports an exported metabase into an existing one.

```
HRESULT Import(
  [unique, in, string] LPCWSTR pszPasswd,
  [unique, in, string] LPCWSTR pszFileName,
  [unique, in, string] LPCWSTR pszSourcePath,
  [unique, in, string] LPCWSTR pszDestPath,
  [in] DWORD dwMDFlags
);
```

pszPasswd: A pointer to a Unicode string containing the password that will be used to decrypt the secure properties of the metabase being imported.

pszFileName: A pointer to a Unicode string containing the name of the file, including directory path, to import settings from. The path SHOULD be local to the server. This file will have been created using the **Export** function, on this server or another server.

pszSourcePath: A pointer to a Unicode string containing the path to the source metabase before import.<5>

pszDestPath: A pointer to a Unicode string containing the path to the new metabase after import.<6>

dwMDFlags: A set of bit flags specifying the export operation to be performed.

Value	Meaning
MD_IMPORT_INHERITED 0x00000001	Inherited settings that were exported because the MD_EXPORT_INHERITED flag was used in creating the data file; will be imported.
MD_IMPORT_NODE_ONLY	Child nodes will not be imported.

Value	Meaning
0x00000002	
MD_IMPORT_MERGE 0x00000004	Imported settings will be merged with any matching existing node settings. When a value for a setting is present in the data file and also in the current metabase, the data file setting will overwrite the existing metabase setting. If you do not set this flag and there is a current node in the metabase that conflicts with the node being imported, the imported node will replace the existing node. All settings from the existing node will be lost regardless of whether the imported node contains the setting or not.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x00000003 HRESULT derived from ERROR_PATH_NOT_FOUND	The system cannot find the file specified.
0x00000006 ERROR_INVALID_HANDLE	The handle is invalid.
0x80070008 HRESULT derived from ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.
0x80070057 HRESULT derived from E_INVALIDARG	One or more arguments are invalid.
0x80070005 HRESULT derived from ERROR_ACCESS_DENIED	Access is denied.
0x800CC801 MD_ERROR_DATA_NOT_FOUND	The specified metadata was not found.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol [MS-RPCE].

The opnum field value for this method is 37.

When processing this call, the server MUST do the following:

- The *pszFileName* MUST exist; otherwise the server MUST return the HRESULT derived from the ERROR_FILE_NOT_FOUND error code.
- The *pszFileName* MUST be local to the server, otherwise the server MUST return the HRESULT derived from the ERROR_PATH_NOT_FOUND error code.
- The server MUST decrypt data with the password the client has provided.
- The source path MUST exist in the exported data; otherwise the server MUST return an ERROR_PATH_NOT_FOUND error code.
- If the destination path exists on the server, the server will replace it and all settings on it with the data from the data file unless the client has passed the MD_IMPORT_MERGE flag.
- If the client has passed the MD_IMPORT_MERGE flag and the destination path exists on the server, the server will overwrite any existing settings with data from the data file, but will keep any settings that are not present in the data file.
- If the MD_IMPORT_INHERITED flag is passed, the server MUST include inherited properties from the data file when importing the data to the server.
- If the MD_IMPORT_NODE_ONLY flag is passed, the server MUST import only the specified node and its settings. Child nodes MUST NOT be included.

3.3.4.5 RestoreHistory (Opnum 38)

The **RestoreHistory** method restores a metabase history entry for a specific history version.

```
HRESULT RestoreHistory(
  [unique, in, string] LPCWSTR pszMDHistoryLocation,
  [in] DWORD dwMDMajorVersion,
  [in] DWORD dwMDMinorVersion,
  [in] DWORD dwMDFlags
);
```

pszMDHistoryLocation: A pointer to a Unicode string containing the absolute path to the location of the history files for the metabase. If an empty string is passed to this function, the server SHOULD use the default history path.<7>

dwMDMajorVersion: An integer value containing the pre-decimal version value of the history entry to restore from. If the *dwMDFlags* contains MD_HISTORY_LATEST, then this value MUST be set to zero.

dwMDMinorVersion: An integer value containing the post-decimal version value of the history entry to restore from. If the *dwMDFlags* contains MD_HISTORY_LATEST, then this value MUST be set to zero.

dwMDFlags: A set of bit flags specifying which options to be executed during the **RestoreHistory** call.

Value	Meaning
MD_HISTORY_LATEST 0x00000001	Restore to the most recent history file. If this is set, the *dwMDMajorVersion* and *dwMDMinorVersion* MUST be set to zero.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.
0x80070002 HRESULT derived from ERROR_FILE_NOT_FOUND	The system cannot find the file specified.
0x80070003 HRESULT derived from ERROR_PATH_NOT_FOUND	The system cannot find the path specified.
0x8007000E E_OUTOFMEMORY	Ran out of memory.
0x80070008 HRESULT derived from ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
0x80070057 HRESULT derived from ERROR_INVALID_PARAMETER	The parameter is incorrect.
0x80070013 HRESULT derived from ERROR_INVALID_DATA	One or more arguments are invalid.
0x800703EC HRESULT derived from ERROR_INVALID_FLAGS	Invalid flags were passed.
0x80070005 HRESULT derived from ERROR_ACCESS_DENIED	Access is denied.
0x800CC802 MD_ERROR_INVALID_VERSION	The version specified in metadata storage was not recognized.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol [MS-RPCE].

The opnum field value for this method is 38.

When processing this call the server MUST do the following:

- The server MUST restore the history from the location passed in the *pszMDHistoryLocation* value. If this location does not exist the server returns the HRESULT derived from the Win32 error code ERROR_PATH_NOT_FOUND.
- If the *pszMDHistoryLocation* value passed in is an empty string, the server uses the default history location as defined by the server specific implementation.

- If the MD_HISTORY_LATEST flag is passed, the server MUST check that the *dwMDMajorVersion* and *dwMDMinorVersion* are 0 and return the HRESULT derived from the Win32 error code E_INVALIDARG.
- If the version requested does not exist, the server MUST return the MD_ERROR_INVALID_VERSION error code.
- If the *dwMDFlags* value contains anything beyond the expected flag values, the server MUST return the ERROR_INVALID_FLAGS error code.
- The server replaces the current metabase data with the data from the history entry specified.

3.3.4.6 RestoreWithPasswd (Opnum 35)

The **RestoreWithPasswd** method restores the metabase from a backup, using a supplied password to decrypt the secure data.

```
HRESULT RestoreWithPasswd(
  [unique, in, string] LPCWSTR pszMDBackupLocation,
  [in] DWORD dwMDVersion,
  [in] DWORD dwMDFlags,
  [unique, in, string] LPCWSTR pszPasswd
);
```

pszMDBackupLocation: A pointer to a Unicode string containing the backup name to be restored. If an empty string is specified the default backup name SHOULD be used.<8>

dwMDVersion: An integer value specifying the version number of the backup to be restored, which MUST be less than or equal to MD_BACKUP_MAX_VERSION (9999) or the following constant.

Value	Meaning
MD_BACKUP_HIGHEST_VERSION 0xFFFFFFFE	Restore from the highest existing backup version in the specified backup name.

dwMDFlags: This parameter is reserved and MUST be set to zero.

pszPasswd: A password string used to decrypt the secure properties in the metabase backup. If the password is not correct, an error is returned. If a password is not supplied, this method functions exactly the same as **Restore** method.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x00000000 S_OK	The call was successful.

Return value/code	Description
0x80070002 HRESULT derived from ERROR_FILE_NOT_FOUND	The system cannot find the file specified.
0x80070003 HRESULT derived from ERROR_PATH_NOT_FOUND	The system cannot find the path specified.
0x80070008 HRESULT derived from ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
0x8007000E E_OUTOFMEMORY	There was not enough memory to complete the method call.
0x8007007B HRESULT derived from ERROR_INVALID_NAME	The file name, directory name, or volume label syntax is incorrect.
0x80070057 E_INVALIDARG	One or more arguments are invalid.
0x80070005 HRESULT derived from ERROR_ACCESS_DENIED	Access is denied.
0x800CC802 MD_ERROR_INVALID_VERSION	The version specified in metadata storage was not recognized.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol [MS-RPCE].

The opnum field value for this method is 35.

When processing this call the server MUST do the following:

- The server restores from a backup that is identified by the *pszMDBackupLocation* parameter and the version number.
- If the *pszMDBackupLocation* parameter is an empty string, the server MUST use the default backup location as defined by the server implementation.
- If the backup location contains illegal characters as defined by the server-specific implementation, the server MUST return the HRESULT derived from the Win32 error code ERROR_INVALID_NAME.
- If the backup location does not exist, the server MUST return the HRESULT derived from the Win32 error code E_INVALIDARG.
- If *dwMDVersion* parameter is greater than MD_BACKUP_MAX_VERSION (9999) and not equal to MD_BACKUP_HIGHEST_VERSION, the server MUST return **HRESULT** derived from the Win32 error code E_INVALIDARG.
- If *dwMDVersion* parameter equals MD_BACKUP_HIGHEST_VERSION, the server MUST restore the metadata to the metadata stored in the highest version at this location.

- Any encrypted data MUST be decrypted with the password the client has provided. If no password is provided, the function behaves exactly as the **Restore** method.
- If the password is not supplied, the server will behave exactly as the **Restore** method.

3.3.5 Timer Events

No protocol timer events are required on the client beyond the timers required in the underlying RPC protocol.

3.3.6 Other Local Events

No local events are maintained on the server other than the events that are maintained in the underlying RPC protocol.

3.4 IMSAdminBase2W Client Role Details

3.4.1 Abstract Data Model

No abstract data model is required.

3.4.2 Timers

No protocol timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3.2.3.2.1.

3.4.3 Initialization

The client MUST perform initialization according to the rules defined in section 3.2.3.

3.4.4 Message Processing Events and Sequencing Rules

The client SHOULD follow the rules defined in section 3.2.4.

3.4.5 Timer Events

No protocol timer events are required on the client beyond the timers required in the underlying RPC protocol.

3.4.6 Other Local Events

A client's invocation of each method is typically the result of local application activity. The local application on the client computer specifies values for all input parameters. No other higher-layer triggered events are processed. The values specified for input parameters are defined in section 2.

No additional local events are used on the client beyond the events maintained in the underlying RPC protocol.

3.5 IMSAdminBase3W Server Role Details

3.5.1 Abstract Data Model

No specific abstract data model is required. This interface uses the same data model as the **IMSAdminBaseW** interface.

3.5.2 Timers

No protocol timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3.2.3.2.1.

3.5.3 Initialization

This protocol uses DCOM initialization, as specified in [MS-DCOM].

3.5.4 Message Processing Events and Sequencing Rules

This DCOM interface inherits the IUnknown interface. Method opnum field values start with 3; opnum values 0 through 2 represent the IUnknown_QueryInterface, IUnknown_AddRef, and IUnknown_Release methods, respectively, as specified in [MS-DCOM].

Methods with opnum field values 3 through 33 are defined in section 3.1.4, and those with field values 34 through 39 are defined in section 3.3.4.

The **IMSAdminBase3W** remote procedure call (RPC) interface extends the **IMSAdminBase2W** interface by providing a method to return the nodes of children from a specified metabase path. The **IMSAdminBase3W** protocol does not maintain client state information; the protocol is stateless.

A Remote Procedure Call (RPC) sequence is a client/server session that includes a security context phase and requests to call remote procedures. For a connection-oriented RPC, the session also includes a binding phase. The RPC client supplies the necessary security information, and for connection-oriented RPCs it also supplies binding information, such as interface name and server endpoint. The sequence of subsequent RPC calls in the session is implementation-specific.

Methods in RPC Opnum Order

Method	Description
GetChildPaths	Returns all child nodes of a specified path from a supplied metadata handle. Opnum: 40

When a remote call is made, the UUID and version number of the interface are specified in the **abstract_syntax** and **abstract_syntax_vers** fields of the incoming RPC_BIND packet, as specified in [MS-RPCE].

3.5.4.1 GetChildPaths (Opnum 40)

The **GetChildPaths** method returns all child nodes of a specified path from a supplied metadata handle.

```
HRESULT GetChildPaths(
  [in] METADATA_HANDLE hMDhandle,
  [unique, in, string] LPCWSTR pszMDPath,
  [in] DWORD cchMDBufferSize,
  [in, out, unique, size_is(cchMDBufferSize)]
    WCHAR* pszBuffer,
  [in, out, unique] DWORD* pcchMDRequiredBufferSize
);
```

hMDhandle: An unsigned 32-bit integer value specifying a handle to a node in the metabase with read permissions as returned by the **OpenKey** method, or the METADATA_MASTER_ROOT_HANDLE.

pszMDPath: A pointer to a Unicode string that contains the path of the node to be opened, relative to *hMDHandle*.

cchMDBufferSize: The size, in **WCHAR**, of the buffer *pszBuffer* to hold the paths for all child nodes under the path specified.

pszBuffer: A pointer to a Unicode character buffer passed in by the caller to store the retrieved child paths. The return data will be a set of **WCHAR** strings, where each includes two NULL-terminating characters.

pcchMDRequiredBufferSize: An integer value indicating the required size of the buffer if the supplied buffer proves to be insufficient. If the supplied buffer is sufficient, this value will not be adjusted.

Return Values: A signed 32-bit value that indicates return status. If the method returns a negative value, it failed. If the 12-bit facility code (bits 16–27) is set to 0x007, the value contains a Win32 error code in the lower 16 bits. Zero or positive values indicate success, with the lower 16 bits in positive non-zero values containing warnings or flags defined in the method implementation. For more information about Win32 error codes and **HRESULT** values, see [MS-ERREF].

Return value/code	Description
0x80070000 S_OK	The call was successful.
0x80070003 HRESULT derived from ERROR_PATH_NOT_FOUND	The system cannot find the path specified.
0x80070057 HRESULT derived from ERROR_INVALID_PARAMETER	The parameter is incorrect.
0x80070005 HRESULT derived from ERROR_ACCESS_DENIED	Access is denied.
0x80070008 HRESULT derived from ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
0x8007000E E_OUT_OF_MEMORY	There was not enough memory to complete the method call.
0x8007007A HRESULT derived from ERROR_INSUFFICIENT_BUFFER	The data area passed to a system call is too small.
0x800700A0 HRESULT derived from ERROR_BAD_ARGUMENTS	One or more arguments are not correct.
0x80004005 E_FAIL	An unspecified error occurred.

Return value/code	Description
0x80070006 E_HANDLE	An invalid handle was passed.
0x800CC800 MD_ERROR_NOT_INITIALIZED	Metadata has not been initialized.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol [MS-RPCE].

The opnum field value for this method is 40.

When processing this call the server MUST do the following:

- The server returns all child paths that are relative to the path provided under the node that is represented by the *hMDHandle* parameter. The server MUST return these as a list of **WCHAR** strings, where each string includes the NULL-terminating character, and the entire list is also followed by a NULL-terminating character.
- The strings returned by the server MUST be compatible with the format used by the **OpenKey** method to open those nodes for retrieving data.
- If the *hMDHandle* parameter is not a valid open handle to the metadata (retrieved by calling **OpenKey**), the server MUST return an E_HANDLE error code.
- If the path requested does not exist in the metadata, the server MUST return the HRESULT derived from the ERROR_PATH_NOT_FOUND error code.
- If the *cchMDBufferSize* is not large enough to contain the child path strings that include the NULL-terminating character, the server MUST return the HRESULT derived from the ERROR_INSUFFICIENT_BUFFER error code, and the server MUST set the *pcchMDRequiredBufferSize* value to the size needed.

3.5.5 Timer Events

No protocol timer events are required on the client beyond the timers required in the underlying RPC protocol.

3.5.6 Other Local Events

No local events are maintained on the server other than the events that are maintained in the underlying RPC protocol.

3.6 IMSAdminBase3W Client Role Details

3.6.1 Abstract Data Model

No abstract data model is required.

3.6.2 Timers

No protocol timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3.2.3.2.1.

3.6.3 Initialization

The client MUST perform initialization according to the rules defined in section 3.5.3.

3.6.4 Message Processing Events and Sequencing Rules

Client SHOULD follow the rules defined in 3.5.4.

3.6.5 Timer Events

No protocol timer events are required on the client beyond the timers required in the underlying RPC protocol.

3.6.6 Other Local Events

A client's invocation of each method is typically the result of local application activity. The local application on the client computer specifies values for all input parameters. No other higher-layer triggered events are processed. The values specified for input parameters are defined in section 2.

No additional local events are used on the client beyond the events maintained in the underlying RPC protocol.

4 Protocol Examples

The following sections provide common scenarios that illustrate the function of the Internet Information Services (IIS) IMSAdminBaseW Remote Protocol.

4.1 General Hookup Example

The following example demonstrates how to get a handle that the DCOM Class Object can use to make the rest of the calls.

- The client initializes COM by calling CoInitializeEx. For more information, see [MSDN-CoInitialize].
- The client initializes COM security by calling CoInitializeSecurity. In this call, the client should set the impersonation level and authentication level that will be used by COM for subsequent calls. The **IMSAdminBaseW** interface does not provide any additional security by itself; it relies on COM for this.<9>
- The client creates an instance of the **IMSAdminBaseW** interface using a CoCreateInstance COM call and passing the remote computer name.
- The client queries the **IMSAdminBaseW** interface from the pointer returned by CoCreateInstance.
- From this point, the client has a valid pointer to the **IMSAdminBaseW** interface that can be used to perform additional processing via the other methods defined on the interface.
- After the client has finished processing, it should release the interface pointer and call CoUninitialize to clear up the COM context.

4.2 BackupWithPasswd Call Example

The client initiates a connection to the server through standard DCOM calls, as specified in [MS-DCOM].

The client calls **BackupWithPasswd** and provides the following parameters:

- A Unicode string that includes the NULL-terminating character, which indicates the backup name. For example, an empty string signifies that the default backup name should be used.
- An integer that indicates the backup version. For example, a parameter value equal to MD_BACKUP_HIGHEST_VERSION signifies that the backup version should be a replacement to the highest existing backup version.
- An integer for backup flags. For example, combining the MD_BACKUP_SAVE_FIRST | MD_BACKUP_OVERWRITE | MD_BACKUP_FORCE_BACKUP flag bits signifies to the server to save non-persisted data before the backup is performed, to continue with the backup even if the attempt to save the non-persisted data fails, and to overwrite existing backups using the same version and name.
- A Unicode string that includes the NULL-terminating character, which is used as a password by the server for encrypting any protected data in the backup.

The client then checks the return code from the function to determine if the backup succeeded.

4.3 EnumHistory Call Example

The client initiates a connection to the server through standard DCOM calls, as specified in [MS-DCOM].

The client sets an index value to zero, which will be used to move through the history entries starting at the first one.

The client calls **EnumHistory** and provides the following parameters:

- An empty buffer with room for 100 **WCHARs**.
- A pointer to a **DWORD**, which the server can fill in with the Major Version Number of the history file being enumerated.
- A pointer to a **DWORD**, which the server can fill in with the Minor Version Number of the history file being enumerated.
- A pointer to a **FILETIME** structure that the server can fill in with the file time of the current history file being enumerated.
- An index value that represents which history file should be enumerated.

If the call is successful, the client reads the default location of the history files from the buffer passed in and processes the rest of the information returned in the two **DWORDs** and the **FILETIME** parameters.

If the call was successful, the client then increments the Index value and makes another call to the **EnumHistory** function to get the next entry.

If the call returned the ERROR_NO_MORE_ITEMS error code, then the client concludes that it has processed all the history entries and has successfully finished.

4.4 Export Call Example

The client initiates a connection to the server through standard DCOM calls, as specified in [MS-DCOM].

The client calls **Export** and provides the following parameters:

- A Unicode string that includes the NULL-terminating character, which is used as a password by the server to encrypt any protected data.
- A Unicode string that includes the NULL-terminating character, and is used by the server as the file into which the data will be exported. The string should represent an existing path on the server.
- A Unicode string that includes the NULL-terminating character, which is used by the server as the configuration store path of the data that the client wants to export from within the store.
- Zero, which is used to inform the server that it can export all child nodes and that it does not have to include inherited properties since those flags are not passed.

The client then checks the return code from the function to determine if the export succeeded.

4.5 Import Call Example

The client initiates a connection to the server through standard DCOM calls, as specified in [MS-DCOM].

The client calls the **Import** method and provides the following parameters:

- A Unicode string that includes the NULL-terminating character, which is used by the server as a password to decrypt any protected data. For example, "MyPassword".
- A Unicode string that includes the NULL-terminating character, which is used by the server as a path to an existing file containing the data to be imported. For example, "d:\\export\\exportedfile.xml".
- A Unicode string that includes the NULL-terminating character, which is used by the server as a configuration path within the file specified in the previous parameter. For example, "/LM/W3SVC/1/". The server uses this configuration path to select the subtree from the configuration data stored in the file.
- A Unicode string that includes the NULL-terminating character, which is used by the server as a configuration path within the configuration store specified in the file. For example, "/LM/W3SVC/901/". The server uses this path to locate the point to where the data should be imported.
- A flag parameter used by the server to determine how the data should be imported. For example, MD_IMPORT_MERGE.

The client then checks the return code from the function to determine if the import succeeded.

4.6 RestoreHistory Call Example

The client initiates a connection to the server through standard DCOM calls, as specified in [MS-DCOM].

The client calls the **RestoreHistory** method and provides the following parameters:

- A Unicode string that includes the NULL-terminating character, which is used by the server to locate history data. For example, an empty string that will signal to the server to restore from the default history location.
- An integer indicating the pre-decimal part of the history version. For example, if the history version is "1234.5678", then this parameter should be 1245. It MUST be zero if the flags parameter is set to MD_HISTORY_LATEST.
- An integer indicating the post-decimal part of the history version. For example, if the history version is "1234.5678", then this parameter should be 5678. It MUST be zero if the flags parameter is set to MD_HISTORY_LATEST.
- An integer indicating to the server how the restore operation should be performed. If this parameter is MD_HISTORY_LATEST, then the server should restore the latest available version of history and ignore data passed as version parameters.

4.7 RestoreWithPasswd Call Example

The client initiates a connection to the server through standard DCOM calls, as specified in [MS-DCOM].

The client calls the **RestoreWithPasswd** method and provides the following parameters:

- A Unicode string that includes the NULL-terminating character, which contains the backup name. For example, an empty string signifies that the default backup name should be used.
- An integer with the backup version. For example, MD_BACKUP_HIGHEST_VERSION signifies that the highest version of backup should be restored.
- A reserved parameter that must be zero.
- A Unicode string that includes the NULL-terminating character, which is used by the server as a password for **decrypting** any protected data in the backup. For example, "MyPassword".

The client then checks the return code from the function to determine if the restore succeeded.

4.8 GetChildPaths Call Example

The client initiates a connection to the server through standard DCOM calls, as specified in [MS-DCOM].

The client uses the **OpenKey** method and provides the following parameters to get a handle to the metabase data:

- METADATA_MASTER_ROOT_HANDLE, which causes the server to open a key relative to the root.
- "/LM/W3SVC", which tells the server to open the lm/w3svc key under the root.
- METADATA_PERMISSION_READ, which asks the server to open the key with read privileges.
- 10, which tells the server to time out after 10 milliseconds if it cannot open the key.
- A pointer to a handle that the server will fill in with the handle to the node that has been opened.

If the **OpenKey** call is successful, the client calls the **GetChildPaths** method, providing the following parameters to determine from the server how much space is required for a successful call to the **GetChildPaths** method:

- The handle to the key opened by the **OpenKey** method.
- An empty string, which is used by the server to locate the child paths relative to the handle passed in the first parameter.
- Zero, which indicates the size of the buffer passed in the next parameter. In this first call to the **GetChildPaths** method, the buffer size is set to zero because the client is attempting to determine the correct size for the buffer.
- NULL. While this parameter is normally used as the buffer to hold the child paths, on this first call to the **GetChildPaths** method the client is only attempting to determine how large the buffer should be, and therefore, this parameter is set to NULL.
- A pointer to a **DWORD**. The server will fill in the **DWORD** with the correct number of bytes to be used as the buffer size in our subsequent call to the **GetChildPaths** method.

The call to the **GetChildPaths** method is expected to return an ERROR_INSUFFICIENT_BUFFER error code, and in the last parameter the number of bytes needed by the buffer in order to hold all of the child paths. If the call to the **GetChildPaths** method fails for any other reason, the client will exit.

Once the client has been informed of the number of bytes needed to hold all of the child paths, it will allocate a buffer of that size.

The client then calls the **GetChildPaths** method again to provide the following parameters:

- The handle to the key opened by the **OpenKey** method.
- An empty string, which is used by the server to locate the child paths relative to the handle passed in the first parameter.
- The number of bytes to allocate for the buffer that will hold the child paths. The number of bytes to allocate was returned by the first call to the **GetChildPaths** method.
- The allocated buffer that will hold the child paths. The size of the buffer is specified in the previous parameter.
- A pointer to a **DWORD**. If the server determines that the number of bytes specified in the third parameter is not sufficient to allocate a buffer large enough to hold all the child paths, then the server will fill in the **DWORD** with the correct number of bytes to allocate for the buffer.

If the child paths were successfully retrieved, the client parses the buffer to locate each child path string. The client searches for an occurrence of double NULL-terminating characters, and when found, processes the child path. The client continues parsing the buffer until the end of the data is reached.

4.9 Reading Sensitive Data from the Server

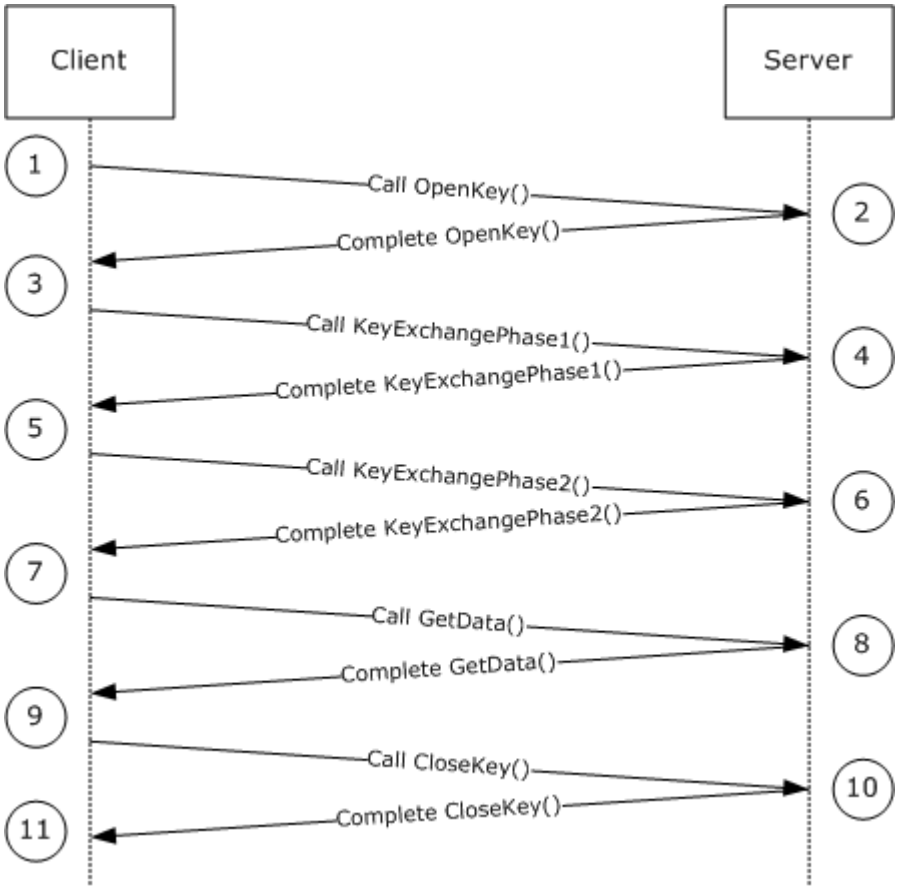


Figure 1: Message sequence for reading sensitive data from the server

The sequence of messages for reading sensitive data from the server is as follows:

1. The client requests that the server open a node. The path location is "/mydata" and METADATA_PERMISSION_READ read access is requested.

2. The server checks if the "/mydata" node exists and whether the connecting user is granted read access, and then returns the handle to the client.

   Before the client sends a request to retrieve specific data from the "/mydata" location, the client will have to negotiate a secure session (that is, one that was not yet negotiated).

3. The client performs phase 1 of the handshake. The client's Key Exchange Key and Signature Key are generated, and Public Keys for both are sent to the server. **Private keys** for both are stored by the client.

4. The server receives the Public Keys from the client and retrieves, or generates/locates, its own server's Key Exchange Key and Signature Key.

   The server also generates the server's Session Key. It encrypts the Session Key with the client's Key Exchange Public Key. The server's Key Exchange Public Key, the server's Signature Public Key, and the encrypted server's Session Key are sent back to the client.

5. The client receives the server's Key Exchange Public Key, the server's Signature Public Key, and the encrypted server's Session Key. It decrypts the server's Session Key using the client's Key Exchange Private Key.

   The client's Session Key gets generated and is encrypted with the server's Key Exchange Public Key. In addition, the server's Session Key Hash is generated based on the client's Session Key, the server's Session Key, and the HASH_TEXT_STRING_1 (see 2.2.2) string constant. The encrypted client's Session Key and the server's Session Key Hash are sent to the server.

6. The server receives the encrypted client's Session Key as well as the server's Session Key Hash from the client. It verifies the hash generated by the client to ensure that the client was able to decrypt the server's Session Key. The server generates the client's Session Key Hash using the client's Session Key and the HASH_TEXT_STRING_2 (see 2.2.2) string constant. The server then sends the client's Session Key Hash to the client.

7. The client receives the client's Session Key Hash from the server. It verifies that the server owns the Private Key for the **Key Exchange Key Pair** and that it was able to decrypt the client's Session Keys.

   By this point in the sequence, the server and the client have exchanged the Session Keys that will be used to encrypt the sensitive data. Also, the Signature Keys have been exchanged which will be used for message integrity checks.

   The client calls the **R_GetData** method to retrieve the sensitive data.

8. The server retrieves the requested data and determines if the METADATA_SECURE secure flag is set. The server encrypts the data value requested, builds the **IIS_CRYPTO_BLOB** message, and sends the data to the client.

9. The client checks the received data and determines if the METADATA_SECURE secure flag is set. The client decrypts the data and verifies the signature.

   The client calls the **CloseKey** method to close the handle that was opened in step 2.

10. The server closes the handle and responds with a success code to the client.

5 Security

The following sections specify security considerations for implementers of the Internet Information Services (IIS) IMSAdminBaseW Remote Protocol.

5.1 Security Considerations for Implementers

Authenticated RPC should be used by this protocol, as specified in C706 section 13.

The IIS IMSAdminBaseW Remote Protocol uses weak keys and cryptographic algorithms. 512-bit RSA keys, 40-bit RC4, and MD5 hash are used to protect sensitive data. For more information, see section 3.1.4.1.1.

The IIS IMSAdminBaseW Remote Protocol includes secure session negotiation, but does not provide support for server side authentication or for handling man-in-the-middle attacks (MITM). For more information, see section 3.1.4.1.1.

The RPC/DCOM packet privacy feature should be used for more robust protection of the data transferred over the IIS IMSAdminBaseW Remote Protocol.<10>

5.2 Index of Security Parameters

Security parameter	Section
Secure session settings (512-bit RSA keys, 40-bit RC4 keys, MD5 hash)	3.1.4.1.1

6 Appendix A: Full IDL

For ease of implementation the full IDL is provided below, where "ms-dtyp.idl" refers to the IDL found in [MS-DTYP] Appendix A and where "ms-dcom.idl" refers to the IDL found in [MS-DCOM] Appendix A.

The syntax uses the IDL syntax extensions defined in [MS-RPCE] sections 2.2.4 and 3.1.1.5.1. For example, as noted in [MS-RPCE] section 2.2.4.9, a pointer_default declaration is not required and pointer_default(unique) is assumed.

```
import "ms-dtyp.idl";
import "ms-dcom.idl";

typedef unsigned long METADATA HANDLE, *PMETADATA HANDLE;

typedef struct  IIS CRYPTO BLOB{
    DWORD BlobSignature;
    DWORD BlobDataLength;
    [size is(BlobDataLength)] unsigned char BlobData[*];
} IIS CRYPTO BLOB;

typedef struct {
    DWORD dwMDIdentifier;
    DWORD dwMDAttributes;
    DWORD dwMDUserType;
    DWORD dwMDDataType;
    DWORD dwMDDataLen;
    [unique, size is(dwMDDataLen)] unsigned char * pbMDData;
    DWORD dwMDDataTag;
} METADATA_RECORD;

typedef struct {
    DWORD dwMDIdentifier;
    DWORD dwMDAttributes;
    DWORD dwMDUserType;
    DWORD dwMDDataType;
    DWORD dwMDDataLen;
    union {
        DWORD dwMDDataOffset;
        unsigned char * pbMDData;
    };
    DWORD dwMDDataTag;
} METADATA GETALL RECORD, *PMETADATA GETALL RECORD;

typedef struct {
    DWORD dwMDPermissions;
    DWORD dwMDSystemChangeNumber;
} METADATA_HANDLE_INFO;

typedef struct {
    [string] WCHAR * pszMDPath;
    DWORD dwMDChangeType;
    DWORD dwMDNumDataIDs;
    [unique, size_is(dwMDNumDataIDs)] DWORD * pdwMDDataIDs;
} MD CHANGE OBJECT W;

#define METADATA MASTER ROOT HANDLE  0
#define ADMINDATA MAX NAME LEN 256
#define MD_BACKUP_MAX_LEN 100

[
```

```
    object,
    uuid(70B51430-B6CA-11d0-B9B9-00A0C922E750),
    pointer_default(unique)
]
interface IMSAdminBaseW : IUnknown
{
    HRESULT AddKey(
        [in] METADATA_HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath);

    HRESULT DeleteKey(
        [in] METADATA HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath);

    HRESULT DeleteChildKeys(
        [in] METADATA HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath);

    HRESULT EnumKeys(
        [in] METADATA_HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [out, size is(ADMINDATA MAX NAME LEN)] LPWSTR pszMDName,
        [in] DWORD dwMDEnumObjectIndex);

    HRESULT CopyKey(
        [in] METADATA_HANDLE hMDSourceHandle,
        [unique, in, string] LPCWSTR pszMDSourcePath,
        [in] METADATA HANDLE hMDDestHandle,
        [unique, in, string] LPCWSTR pszMDDestPath,
        [in] BOOL bMDOverwriteFlag,
        [in] BOOL bMDCopyFlag);

    HRESULT RenameKey(
        [in] METADATA HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [unique, in, string] LPCWSTR pszMDNewName);

    HRESULT R_SetData(
        [in] METADATA_HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [in] METADATA RECORD * pmdrMDData);

    HRESULT R_GetData(
        [in] METADATA_HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [in, out] METADATA RECORD * pmdrMDData,
        [out] DWORD *pdwMDRequiredDataLen,
        [out] IIS CRYPTO BLOB **ppDataBlob);

    HRESULT DeleteData(
        [in] METADATA HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [in] DWORD  dwMDIdentifier,
        [in] DWORD  dwMDDataType);

    HRESULT R_EnumData(
        [in] METADATA HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [in, out] METADATA RECORD * pmdrMDData,
        [in] DWORD dwMDEnumDataIndex,
        [out] DWORD *pdwMDRequiredDataLen,
        [out] IIS_CRYPTO_BLOB **ppDataBlob);

    HRESULT R GetAllData(
        [in] METADATA_HANDLE hMDHandle,
```

```
    [unique, in, string] LPCWSTR pszMDPath,
    [in] DWORD dwMDAttributes,
    [in] DWORD dwMDUserType,
    [in] DWORD dwMDDataType,
    [out] DWORD *pdwMDNumDataEntries,
    [out] DWORD *pdwMDDataSetNumber,
    [in] DWORD dwMDBufferSize,
    [out] DWORD *pdwMDRequiredBufferSize,
    [out] IIS_CRYPTO_BLOB **ppDataBlob);

HRESULT DeleteAllData(
    [in] METADATA HANDLE hMDHandle,
    [unique, in, string] LPCWSTR pszMDPath,
    [in] DWORD dwMDUserType,
    [in] DWORD dwMDDataType);

HRESULT CopyData(
    [in] METADATA HANDLE hMDSourceHandle,
    [unique, in, string] LPCWSTR pszMDSourcePath,
    [in] METADATA_HANDLE hMDDestHandle,
    [unique, in, string] LPCWSTR pszMDDestPath,
    [in] DWORD dwMDAttributes,
    [in] DWORD dwMDUserType,
    [in] DWORD dwMDDataType,
    [in] BOOL bMDCopyFlag);

HRESULT GetDataPaths(
    [in] METADATA HANDLE hMDHandle,
    [unique, in, string] LPCWSTR pszMDPath,
    [in] DWORD dwMDIdentifier,
    [in] DWORD dwMDDataType,
    [in] DWORD dwMDBufferSize,
    [out, size_is(dwMDBufferSize)] WCHAR *pszBuffer,
    [out] DWORD *pdwMDRequiredBufferSize);

HRESULT OpenKey([in] METADATA HANDLE hMDHandle,
    [unique, in, string] LPCWSTR pszMDPath,
    [in] DWORD  dwMDAccessRequested,
    [in] DWORD  dwMDTimeOut,
    [out] METADATA HANDLE * phMDNewHandle);

HRESULT CloseKey(
    [in] METADATA_HANDLE hMDHandle);

HRESULT ChangePermissions([in] METADATA HANDLE hMDHandle,
    [in] DWORD  dwMDTimeOut,
    [in] DWORD  dwMDAccessRequested);

HRESULT SaveData();

HRESULT GetHandleInfo([in] METADATA HANDLE hMDHandle,
    [out] METADATA HANDLE INFO * pmdhiInfo);

HRESULT GetSystemChangeNumber(
    [out] DWORD *pdwSystemChangeNumber);

HRESULT GetDataSetNumber([in] METADATA HANDLE hMDHandle,
    [unique, in, string] LPCWSTR pszMDPath,
    [out] DWORD *pdwMDDataSetNumber);

HRESULT SetLastChangeTime([in] METADATA HANDLE hMDHandle,
    [unique, in, string] LPCWSTR pszMDPath,
    [in] PFILETIME  pftMDLastChangeTime,
    [in] BOOL bLocalTime);
```

```
    HRESULT GetLastChangeTime([in] METADATA HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [out] PFILETIME  pftMDLastChangeTime,
        [in] BOOL bLocalTime);

    HRESULT R KeyExchangePhase1(
        [unique, in] IIS CRYPTO BLOB *pClientKeyExchangeKeyBlob,
        [unique, in] IIS_CRYPTO_BLOB *pClientSignatureKeyBlob,
        [out] IIS_CRYPTO_BLOB **ppServerKeyExchangeKeyBlob,
        [out] IIS CRYPTO BLOB **ppServerSignatureKeyBlob,
        [out] IIS CRYPTO BLOB **ppServerSessionKeyBlob);

    HRESULT R KeyExchangePhase2(
        [unique, in] IIS_CRYPTO_BLOB *pClientSessionKeyBlob,
        [unique, in] IIS_CRYPTO_BLOB *pClientHashBlob,
        [out] IIS CRYPTO BLOB **ppServerHashBlob);

    HRESULT Backup(
        [unique, in, string] LPCWSTR pszMDBackupLocation,
        [in] DWORD dwMDVersion,
        [in] DWORD dwMDFlags);

    HRESULT Restore(
        [unique, in, string] LPCWSTR pszMDBackupLocation,
        [in] DWORD dwMDVersion,
        [in] DWORD dwMDFlags);

    HRESULT EnumBackups(
        [in, out, size is(MD BACKUP MAX LEN)] LPWSTR pszMDBackupLocation,
        [out] DWORD *pdwMDVersion,
        [out] PFILETIME  pftMDBackupTime,
        [in]  DWORD dwMDEnumIndex);

    HRESULT DeleteBackup(
        [unique, in, string] LPCWSTR pszMDBackupLocation,
        [in] DWORD dwMDVersion);

    HRESULT UnmarshalInterface(
        [out] IMSAdminBaseW **piadmbwInterface);

    HRESULT R GetServerGuid(
        [out] GUID *pServerGuid);
};

[
    object,
    uuid(8298d101-f992-43b7-8eca-5052d885b995),
    pointer default(unique)
]
interface IMSAdminBase2W : IMSAdminBaseW
{

  HRESULT BackupWithPasswd(
      [unique, in, string] LPCWSTR pszMDBackupLocation,
      [in] DWORD dwMDVersion,
      [in] DWORD dwMDFlags,
      [unique, in, string] LPCWSTR pszPasswd);

  HRESULT RestoreWithPasswd(
      [unique, in, string] LPCWSTR pszMDBackupLocation,
      [in] DWORD dwMDVersion,
      [in] DWORD dwMDFlags,
      [unique, in, string] LPCWSTR pszPasswd);

  HRESULT Export(
```

```
        [unique, in, string] LPCWSTR pszPasswd,
        [unique, in, string] LPCWSTR pszFileName,
        [unique, in, string] LPCWSTR pszSourcePath,
        [in] DWORD dwMDFlags);

    HRESULT Import(
        [unique, in, string] LPCWSTR pszPasswd,
        [unique, in, string] LPCWSTR pszFileName,
        [unique, in, string] LPCWSTR pszSourcePath,
        [unique, in, string] LPCWSTR pszDestPath,
        [in] DWORD dwMDFlags);

    HRESULT RestoreHistory(
        [unique, in, string] LPCWSTR pszMDHistoryLocation,
        [in] DWORD dwMDMajorVersion,
        [in] DWORD dwMDMinorVersion,
        [in] DWORD dwMDFlags);

    HRESULT EnumHistory(
        [in, out, size_is(ADMINDATA_MAX_NAME_LEN)]
             LPWSTR  pszMDHistoryLocation,
        [out] DWORD *pdwMDMajorVersion,
        [out] DWORD *pdwMDMinorVersion,
        [out] PFILETIME  pftMDHistoryTime,
        [in]  DWORD dwMDEnumIndex);
};

[
    object,
    uuid(f612954d-3b0b-4c56-9563-227b7be624b4),
    pointer default(unique)
]
interface IMSAdminBase3W : IMSAdminBase2W
{
    HRESULT GetChildPaths(
        [in] METADATA HANDLE hMDHandle,
        [unique, in, string] LPCWSTR pszMDPath,
        [in] DWORD cchMDBufferSize,
        [in, out, unique, size_is(cchMDBufferSize)]
             WCHAR * pszBuffer,
        [in, out, unique]
             DWORD * pcchMDRequiredBufferSize);
};
```

7 Appendix B: Windows Behavior

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

- Windows Server 2008
- Windows Vista
- Windows Server 2003
- Windows XP
- Windows 2000
- Windows NT 4.0

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

<1> Section 3.1.4.1: Transferring sensitive data without IMSAdminBaseW Remote Protocol-level encryption.

Windows NT supports a mode where client and server may exchange sensitive data (see section 3.1.4.1) over the IMSA protocol without having a valid secure session negotiated. That mode applies to only machines with the French locale.

To negotiate cleartext mode of operation client and server still MUST go through the secure session negotiation. They have to handle **R_KeyExchangePhase1** and **R_KeyExchangePhase2**, but with the following changes:

- Any Key Exchange Public Key BLOB is replaced with a **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to the CLEARTEXT_BLOB_SIGNATURE signature, and where the **BlobData** field contains the "KeYk" string without the terminating NULL character.
- Any Signature Public Key BLOB is replaced with a **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to the CLEARTEXT_BLOB_SIGNATURE signature, and where the **BlobData** field contains the "SiGk" string without the terminating NULL character.
- Any Session Key BLOB is replaced with a **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to the CLEARTEXT_BLOB_SIGNATURE signature, and where the **BlobData** field contains the "SeSk" string without the terminating NULL character.
- Any Hash exchanged is replaced with a **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to the CLEARTEXT_BLOB_SIGNATURE signature, and where the **BlobData** field contains one byte set to 0x00.

Sensitive data will not be encrypted in this mode of operation. Instead of using a **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to ENCRYPTED_DATA_SIGNATURE, the sensitive data will be placed into a **IIS_CRYPTO_BLOB** structure with the **BlobSignature** field set to CLEARTEXT_DATA_SIGNATURE in a call to **R_SetData**, **R_GetData**, **R_EnumData**, and **R_GetAllData**.

Decryption does not apply in this mode of operation. Instead of decrypting data store in a **IIS_CRYPTO_BLOB** structure, the data is simply retrieved from the **IIS_CRYPTO_BLOB** instance with a CLEARTEXT_DATA_SIGNATURE signature.

<2> Section 3.1.4.1: Windows Server 2003, Windows Vista, and Windows Server 2008 take advantage of the RPC/COM packet privacy feature. It provides a protective layer over the weak encryption used to protect data that is part of the IMSAdminBaseW Remote protocol. Note that RPC/COM packet privacy is not a replacement of the IMSAdminBaseW Remote protocol security features.

<3> Section 3.3.4.2: Default path is %windir%\system32\inetsrv\history.

<4> Section 3.3.4.3: Path example is "/lm/w3svc/1".

<5> Section 3.3.4.4: Path example is "/lm/w3svc/1".

<6> Section 3.3.4.4: Path example is "/lm/w3svc/999".

<7> Section 3.3.4.5: The default history path on Windows Server 2003 is "%windir%\system32\inetsrv\history".

<8> Section 3.3.4.6: The default backup name for all versions of Windows is "MDBackupUp". The location of backups is "%windir%\system32\inetsrv\metaback" directory.

<9> Section 4.1: A Windows implementation of this protocol requires the RPC_C_IMP_LEVEL_IMPERSONATE impersonation level to be set.

<10> Section 5.1: Windows Server 2003, Windows Vista, and Windows Server 2008 take advantage of the RPC/COM packet privacy feature RPC_C_AUTHN_LEVEL_PKT_PRIVACY. This feature provides a protective layer over the weak encryption, as described in section 3.1.4.1.1.

8 Index

L

M

N

O

P

R

S

T

U

V

W