[MS-EFSR]:
Encrypting File System Remote (EFSRPC) Protocol Specification

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

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

Revision Summary

Date	Revision History	Revision Class	Comments
10/22/2006	0.01		MCPP Milestone 1 Initial Availability
01/19/2007	1.0		MCPP Milestone 1
03/02/2007	1.1		Monthly release
04/03/2007	1.2		Monthly release

Date	Revision History	Revision Class	Comments
05/11/2007	1.3		Monthly release
06/01/2007	1.3.1	Editorial	Revised and edited the technical content.
07/03/2007	1.3.2	Editorial	Revised and edited the technical content.
07/20/2007	1.3.3	Editorial	Revised and edited the technical content.
08/10/2007	1.3.4	Editorial	Revised and edited the technical content.
09/28/2007	2.0	Major	Updated and revised the technical content.
10/23/2007	2.1	Minor	Updated the technical content.
11/30/2007	2.1.1	Editorial	Revised and edited the technical content.
01/25/2008	2.1.2	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

The Encrypting File System Remote (EFSRPC) Protocol is a Microsoft-proprietary protocol that is used for performing maintenance and management operations on encrypted data that is stored remotely and accessed over a network. It is used in Windows to manage **files** that reside on remote file servers and are encrypted using the **Encrypting File System (EFS)**.

1.1 Glossary

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

Access Control List (ACL)
Binary Large Object (BLOB)
Binding
Certificate
Decrypting
Domain
Encrypting File System (EFS)
Encryption
Endpoint
File System
Flags
Globally Unique Identifier (GUID)
Key
Named Pipe
Opnum
Plaintext
Private Key
Public Key
Public Key Infrastructure (PKI)
Remote Procedure Call (RPC)
Rivest-Shamir-Adleman (RSA)
RPC Protocol Sequence
RPC Transport
Security Identifier (SID)
Security Provider
Security Support Provider Interface (SSPI)
Server
Server Message Block (SMB)
Stream
Unicode
Universal Naming Convention (UNC)
Universally Unique Identifier (UUID)
Well-Known Endpoint
X.509

The following terms are specific to this document:

Advanced Encryption Standard (AES): A cryptographic algorithm that can be used to protect electronic data. The AES algorithm can be used to encrypt (encipher) and **decrypt** (decipher) information. **Encryption** converts data to an unintelligible form called ciphertext; **decrypting** the ciphertext converts the data back into its original form, called **plaintext**. AES is a symmetric cipher, meaning that the same **key** is used for the **encryption** and decryption operations. It is also a block cipher, meaning that it operates on fixed-size blocks of **plaintext**

and ciphertext, and requires the size of the **plaintext** as well as the ciphertext to be an exact multiple of this block size. AES is specified in [FIPS197].

Data Decryption Field (DDF): The portion of the EFSRPC Metadata that contains information that enables authorized users to **decrypt** the **file**.

Data Recovery Agent (DRA): A logical entity corresponding to an asymmetric key pair that is configured as part of administrative policy by an administrator. When an EFS **file** is created or modified, it is also automatically configured to give all DRAs in effect at that time the ability to **decrypt** it.

Data Recovery Field (DRF): The portion of the EFSRPC Metadata that contains information that enables authorized **DRAs** to **decrypt** the **file**.

EFS Raw Data Format: The data format used by the EFSRPC raw methods to marshal the contents and metadata of an encrypted **file** into a single-bit **stream**. It is specified in section 2.2.3.

EFSRPC Metadata: The additional data stored with an encrypted **file** to enable authorized users to access the data in the **file**. The format of this metadata is implementation-dependent. The EFSRPC Metadata general requirements are specified in detail in section 2.2.2 and the Windows format is specified in associated endnotes in Appendix B of this specification..

File: A unit of data in the **file system**. An encrypted file consists of encrypted data along with the metadata required for a user to **decrypt** the file. The file and its metadata are protected using **public key** cryptography such that an authorized user's **private key** is required to **decrypt** the file.

File Encryption Key (FEK): The symmetric key that is used to encrypt the data in an EFS-protected **file**. The FEK is further encrypted and stored in the **file** metadata such that only authorized users can access it.

Folder: A container for **files** and other folders. A folder may be encrypted. The semantics of encrypting a folder are implementation-dependent. In the Windows implementation, encrypting a folder does not directly cause any data to be encrypted. Encrypting a folder in Windows has the following consequences:

- EFSRPC Metadata is created and stored with the folder.
- An **NTFS** attribute is set on the folder to signify that it is encrypted. **NTFS** checks this attribute when any new **files** or folders are created in the folder. **NTFS** will automatically encrypt any **files** or folders created within a folder that has this attribute set.

New Technology File System (NTFS): The native **file system** of Windows 2000 and later versions of the Windows operating system. Within this document, this term is occasionally used to refer to the operating system subsystem that implements NTFS support. For more information, see [MSFT-NTFS].

Sparse File: A **file** containing large sections of data composed only of zeros, which is marked as such in the **NTFS**. The **file system** saves disk space by only allocating as many ranges on disk as are required to completely reconstruct the non-zero data. When an attempt is made to read in the non-allocated portions of the **file** (also known as holes), the **file system** automatically returns zeros to the caller.

Valid Data Length (VDL): In **NTFS**, there are two important concepts of **file** length: the end-of-file (EOF) marker and the valid data length (VDL). The EOF indicates the actual length of

the **file**. The VDL identifies the length of valid data on disk. Any reads between VDL and EOF automatically return zeros.

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-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.

[MS-SMB] Microsoft Corporation, "Server Message Block (SMB) Protocol Specification", July 2007.

[MS-SMB2] Microsoft Corporation, "Server Message Block (SMB) Version 2.0 Protocol Specification", July 2007.

[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

[UNICODE] The Unicode Consortium, "Unicode Home Page", 2006, http://www.unicode.org/

1.2.2 Informative References

[CIFS] Leach, P. and Naik, D., "A Common Internet File System (CIFS/1.0) Protocol", March 1997, http://www.microsoft.com/about/legal/intellectualproperty/protocols/BSTD/CIFS/dr aft-leach-cifs-v1-spec-02.txt

If you have any trouble finding [CIFS], please check here.

[FIPS180] Federal Information Processing Standards Publication, "Secure Hash Standard", FIPS PUB 180-1, April 1995, http://www.itl.nist.gov/fipspubs/fip180-1.htm

[FIPS180-2] Federal Information Processing Standards Publication, "Secure Hash Standard", FIPS PUB 180-2, August 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[FIPS197] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 197: Advanced Encryption Standard (AES)", November 2001, http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf

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

[MIDLINF] Microsoft Corporation, "MIDL Language Reference", http://msdn2.microsoft.com/en-us/library/aa367088.aspx

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

[MSDN-CSPCTX] Microsoft Corporation, "Cryptographic Service Provider Contexts", http://msdn2.microsoft.com/en-us/library/Aa380246

[MSDN-CSPR] Microsoft Corporation, "Cryptographic Service Providers", http://msdn2.microsoft.com/en-us/library/aa380245.aspx

[MSDN-RPCTSEC] Microsoft Corporation, "Using Transport-Level Security on the Client", http://msdn2.microsoft.com/en-us/library/aa379194.aspx

[MSFT-EFS] Bragg, R., "The Encrypting File System", http://www.microsoft.com/technet/security/topics/cryptographyetc/efs.mspx

[MSFT-NTFS] Microsoft Corporation, "NTFS Technical Reference", March 2003, http://technet2.microsoft.com/WindowsServer/en/Library/81cc8a8a-bd32-4786-a849-03245d68d8e41033.mspx

[MSFT-XPUEFS] Microsoft Corporation, "Windows XP Professional Resource Kit: Using Encrypting File System", November 2005, http://www.microsoft.com/technet/prodtechnol/winxppro/reskit/c18621675.mspx

[MS-SECO] Microsoft Corporation, "Windows Security Overview", January 2007.

[MS-WDV] Microsoft Corporation, "Web Distributed Authoring and Versioning (WebDAV) Protocol: Client Extensions", August 2007.

[PIPE] Microsoft Corporation, "Named Pipes", http://msdn2.microsoft.com/en-us/library/aa365590.aspx

[RFC2437] Kaliski, B., Staddon, J., "PKCS #1: RSA Cryptography Specifications Version 2.0", RFC 2437, October 1998, http://www.ietf.org/rfc/rfc2437.txt

[RFC3280] Housley, R., Polk, W., Ford, W., and Solo, D., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3280, April 2002, http://www.ietf.org/rfc/rfc3280.txt

[TDEA] National Institute of Standards and Technology, "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", Special Publication 800-67, May 2004.

[X509] ITU-T, "Information Technology - Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks", Recommendation X.509, August 2005, http://www.itu.int/rec/T-REC-X.509/en

Note There is a charge to download the specification.

[X690] ITU-T, "Information Technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", Recommendation X.690, July 2002, http://www.itu.int/rec/T-REC-X.690/en

Note There is a charge to download the specification.

1.3 Protocol Overview (Synopsis)

The Encrypting File System Remote Protocol (hereafter referred to as EFSRPC) is a **Remote Procedure Call (RPC)** interface that is used to manage data objects stored in an encrypted form. The objective of encrypting data in this fashion is to enforce access control policies and to provide confidentiality from unauthorized users.

EFSRPC is implemented in Windows to provide remote management for files encrypted by the Encrypting File System (EFS). EFS is the ability of the **NTFS** file system to encrypt files on disk in a manner that is transparent to the user. For more information on EFS, see [MSFT-EFS]. For more information about NTFS, see [MSFT-NTFS].

EFSRPC does not address how data is encrypted, how the encrypted data is stored, or how it is accessed for routine operations such as reading, writing, creating, and deleting. All these actions are specific to the server implementation. On Windows, NTFS provides the storage mechanism (the file is the unit of storage) and the Server Message Block (SMB) Protocol provides remote access to such files. For more information about **SMB**, see [MS-SMB] and [MS-SMB2].

EFSRPC models the underlying data **encryption** architecture using two basic constructs:

- A set of data objects, each of which is encrypted independently and can be managed independently.
- A set of access control subjects, each of which is represented by a key pair generated by a **public key** cryptographic algorithm. The public key of this key pair is embedded in a **certificate** and may be widely distributed in that form. The corresponding **private key** is held solely by the user or users who represent that subject. Thus, a given access control subject may correspond to one or more users, and a given user may possess the private keys for zero or more access control subjects. Access control subjects are further divided into two types:
  - Unprivileged user subjects, which are used for routine data access by ordinary users of the system. For convenience, this specification refers to such subjects as user certificate.
  - **Data Recovery Agents (DRAs)**, which are controlled by system administrators. The storage system ensures that all active DRAs for the system are automatically authorized to access all encrypted objects on the system. If an unprivileged user loses the private key, an administrator can use a DRA's private key to recover the contents of encrypted objects.

EFSRPC also assumes that each encrypted object is associated with some security-related metadata, which contains information required for authorized users and DRAs to access the **plaintext** of the object. This specification refers to this as the EFSRPC Metadata.

EFSRPC does not specify how data is encrypted, stored, or accessed. It is possible to build a compliant EFSRPC implementation that uses a mechanism, such as **access control lists (ACLs)**, instead of encryption to control access to data objects. For the purposes of this specification, the term encrypted is used to indicate that a data object and its metadata can be successfully manipulated through the EFSRPC methods, with the exception of the **EfsRpcEncryptFileSrv** method, which converts data objects from an unencrypted state to an encrypted state.<1>

Within the above model, EFSRPC provides the following categories of management routines. The syntax of the individual methods and rules for how these methods are processed on the server are specified in section 3.1.4.

- Requesting the server to convert objects from encrypted state to unencrypted state and vice versa. Methods:

- **EfsRpcEncryptFileSrv (section 3.1.4.5)**
- **EfsRpcDecryptFileSrv (section 3.1.4.6)**

- Creating, querying, and manipulating the EFSRPC Metadata. Clients use the following methods to query and change which user certificates can be used to **decrypt** an encrypted object. The set of user certificates with access to an object needs to be changed when the set of users with access to the object changes or when a user with access to the object changes the user certificate. The following methods can also be used to copy the access rights from one object to another; the **EfsRpcDuplicateEncryptionInfoFile**, **EfsRpcGetEncryptedfileMetadata**, and **EfsRpcSetEncryptedFileMetadata** methods are particularly well-suited for this purpose. The **EfsRpcGetEncryptedfileMetadata** and **EfsRpcSetEncryptedFileMetadata** methods expose the entire EFSRPC Metadata to the client, and the client may choose to manipulate this metadata if desired. These methods can also be used by a client that is not capable of manipulating the server's EFSRPC Metadata format, and, in this case, other methods may be used to manipulate metadata.
  - **EfsRpcQueryUsersOnFile (section 3.1.4.7)**
  - **EfsRpcQueryRecoveryAgents (section 3.1.4.8)**
  - **EfsRpcRemoveUsersFromFile (section 3.1.4.9)**
  - **EfsRpcAddUsersToFile (section 3.1.4.10)**
  - **EfsRpcFileKeyInfo (section 3.1.4.12)**
  - **EfsRpcDuplicateEncryptionInfoFile (section 3.1.4.13)**
  - **EfsRpcAddUsersToFileEx (section 3.1.4.14)**
  - **EfsRpcFileKeyInfoEx (section 3.1.4.15)**
  - **EfsRpcGetEncryptedFileMetadata (section 3.1.4.16)**
  - **EfsRpcSetEncryptedFileMetadata (section 3.1.4.17)**
- Performing backup of encrypted objects in ciphertext form along with their EFSRPC Metadata, and restoring encrypted objects from such backups. Depending on the implementation of these methods, the backups that are created may expose the implementation-specific EFSRPC Metadata format to the client. The Windows implementation of these methods exposes the Windows EFSRPC Metadata format; however, Windows applications do not manipulate this information. The following methods are suitable for secure content archival or transferring encrypted data securely between servers of the same implementation because they do not require decrypting the data.
  - **EfsRpcOpenFileRaw (section 3.1.4.1)**
  - **EfsRpcReadFileRaw (section 3.1.4.2)**
  - **EfsRpcWriteFileRaw (section 3.1.4.3)**
  - **EfsRpcCloseRaw (section 3.1.4.4)**
- Controlling the server's encryption subsystem. Methods:
  - **EfsRpcFlushEfsCache (section 3.1.4.18)**

Most of the EFSRPC routines are stateless and can be called in any order. When one of these routines is called, the message exchange is as follows:

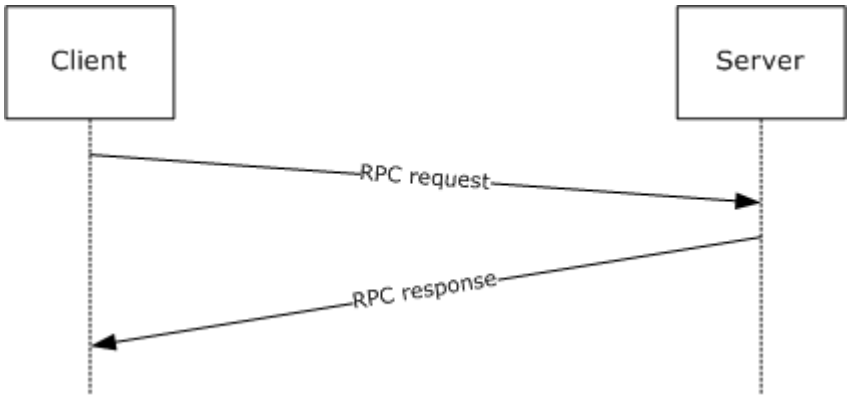


Figure 1: Message exchange for stateless routines

The exceptions to the above are the **EfsRpcOpenFileRaw**, **EfsRpcReadFileRaw**, **EfsRpcWriteFileRaw**, and **EfsRpcCloseRaw** calls, collectively known as the EFSRPC raw methods, which needs to be called in a specific order. The following two sequences are permissible:

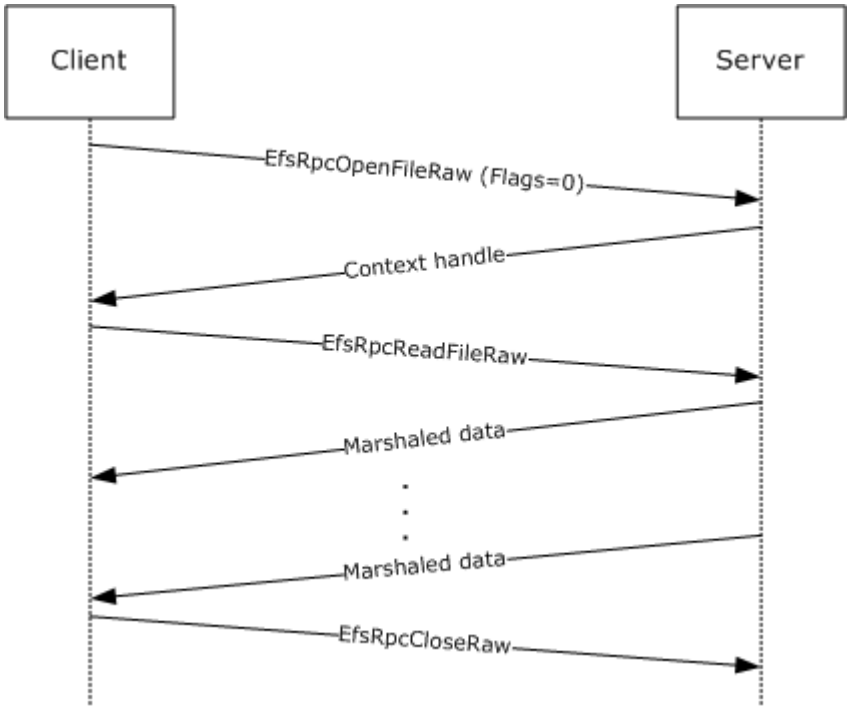


Figure 2: Message sequence for opening a file

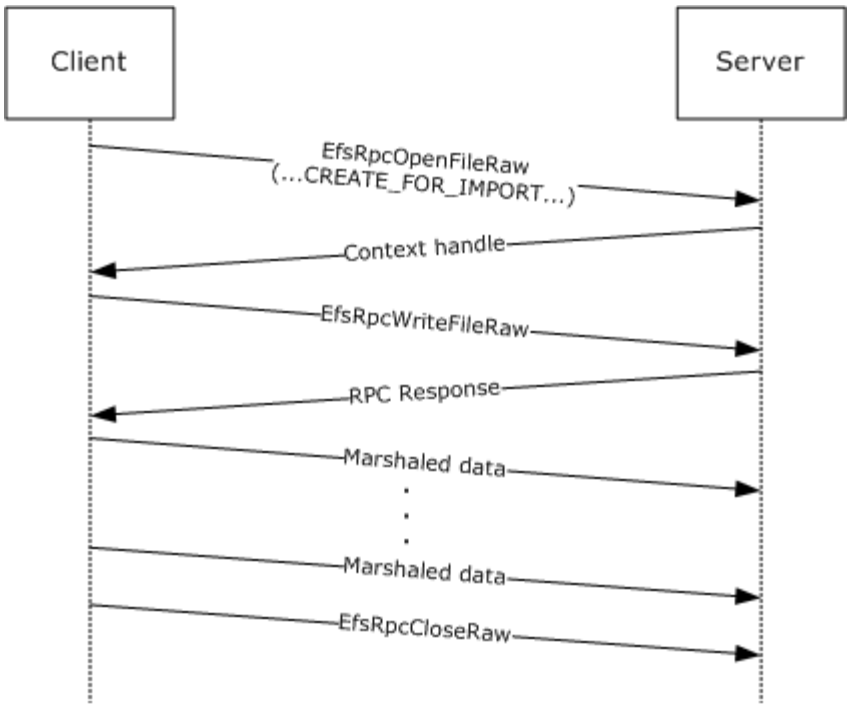


Figure 3: Message sequence for importing a file

1.4 Relationship to Other Protocols

The Encrypting File System Remote Protocol is built on the Microsoft Remote Procedure Call (RPC) interface (as specified in [C706] and [MS-RPCE]). EFSRPC uses the Server Message Block (SMB) Protocol [MS-SMB] [MS-SMB2] as its **RPC transport**. Specifically, it uses **named pipes** over SMB (that is, **RPC protocol sequence** ncacn_np) as its transport mechanism. Either version 1 or version 2 of SMB may be used. The client has to connect to the server over SMB and negotiate a version of SMB before it can access the named pipe that is the RPC **endpoint** on the server.

Windows also supports the storage of encrypted files via WebDAV [MS-WDV]. However, this feature does not use EFSRPC. This feature does not alter the WebDAV Protocol. Windows clients store encrypted files on WebDAV servers in the EFSRPC Raw Data Format, but the Windows WebDAV client performs all encryption and decryption operations locally. It also performs the local operations necessary to transform the file to and from the EFSRPC Raw Data Format during upload and download respectively. For more information, see [MSFT-XPUEFS].

1.5 Prerequisites/Preconditions

To use EFSRPC with a remote server, the client is required to possess valid credentials recognized by the server and be able to pass authentication and authorization checks for access to the encrypted data on the server. If secure operation is desired, the server is required to register an appropriate server principal name/authentication service pair that supports a protection level that provides packet integrity. Additionally, the client must be configured to associate the appropriate server principal name and authentication, and authorization and protection level with its **binding** when connecting to the server.<2>

1.6 Applicability Statement

This protocol is appropriate for remotely managing encrypted data objects on a server. It is used by Windows clients to manage EFSRPC-protected files on remote file servers using either version 1 or version 2 of the SMB Protocol. It does not specify any particular data protection mechanism.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas.

Supported Transports: This protocol uses RPC for communication. It uses named pipes as the transport mechanism, as specified in section 2.1.

Protocol Versions: The RPC runtime negotiates the version of the EFSRPC interface, as specified in [C706]. The only supported version of this protocol is 1.0, as specified in section 3.1.4.

Security and Authentication Methods: EFSRPC does not specify any methods for authenticating access to the objects it operates on. The underlying data encryption and storage system may implement any authentication mechanism. In Windows, such authentication is provided by SMB, as specified in [MS-SMB] and [MS-SMB2]. An EFSRPC server may register a server principal name/authentication service pair to enable secure RPC communications, and a client may choose to associate this security service with its binding when connecting to the server, as specified in section 3.

Capability Negotiation: Implicit negotiation of RPC security mechanisms may be performed through the security-related APIs specified in [C706] Chapter 13. The security mechanisms negotiated by Windows clients and servers are as specified in section 2.1.

1.8 Vendor-Extensible Fields

EFSRPC does not include any vendor-extensible fields.

This protocol uses Win32 error codes. These values are taken from the Windows error number space as specified in [MS-ERREF] section 3. Vendors SHOULD<3> reuse those values with their indicated meaning. Using any other value runs the risk of a collision in the future.

1.9 Standards Assignments

Parameter	Value
RPC Well-Known Endpoint	\pipe\lsarpc
RPC Interface UUID	{c681d488-d850-11d0-8c52-00c04fd90f7e}

2 Messages

2.1 Transport

The client and **server** MUST communicate over RPC, using named pipes over the Server Message Block (SMB) Protocol. The SMB version, capabilities, and authentication used for this connection are negotiated between the client and server when the connection is established, as specified in [MS-SMB] and [MS-SMB2].

EFSRPC messages to remote servers MUST be sent using the **well-known endpoint** \pipe\lsarpc. The server interface is identified by **UUID** [c681d488-d850-11d0-8c52-00c04fd90f7e], version 1.0.

The EFSRPC client MUST use explicit binding to create the RPC binding handle used to connect to the server, unless otherwise specified in section 3.1.4.

A server SHOULD<4> register one or more server principal name/authentication service pairs that provide a protection level that includes packet integrity. A client SHOULD<5> attempt to associate suitable security information with its binding for the EFSRPC raw methods. A client MAY also attempt to negotiate the use of RPC security for calls to other EFSRPC methods.

2.2 Common Data Types

This section specifies the syntax of EFSRPC data types. In addition to the RPC base types and definitions specified in [C706] and [MS-DTYP], the additional data types given below are defined in the Microsoft Interface Definition Language (MIDL) specification for this RPC interface. This protocol MUST indicate to the RPC runtime that it is to support the NDR20 transfer syntax only, as specified in [C706] Part 4.

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

2.2.1 EFSRPC Identifiers

An EFSRPC identifier is used to uniquely refer to an encrypted data object on a remote server. The format of the identifier used is implementation-specific. It MUST be represented as a null-terminated **Unicode** string in UTF-16 encoding. The server MUST return an error if it is passed an identifier that violates the syntactic rules imposed by its implementation.<6>

2.2.2 EFSRPC Metadata

The EFSRPC Metadata<7> is attached to an encrypted object and contains information required to decrypt it. It is used explicitly by the **EfsRpcGetEncryptedFileMetadata** (3.1.4.16) and **EfsRpcSetEncryptedFileMetadata** (3.1.4.17) methods. The EFSRPC Metadata is also used implicitly by the EFSRPC raw methods, because it forms part of the EFSRPC Raw Data Format.

The structure of the EFSRPC Metadata is implementation-dependent. An EFSRPC server SHOULD return an error if EFSRPC Metadata is passed to it in an unsupported format.

2.2.3 EFSRPC Raw Data Format

The EFSRPC raw data format<8> is used by the EFSRPC raw methods. The output of the **EfsRpcReadFileRaw** method MUST conform to this format. The input to the **EfsRpcWriteFileRaw** method MUST conform to the EFSRPC Raw Data Format. The details of this format are implementation-dependent. An EFSRPC server MUST validate input data passed to it by the

EfsRpcWriteFileRaw method, and SHOULD return an error if this data is in an unsupported format. Instead of returning an error, implementations MAY also choose to abort the **EfsRpcWriteFileRaw** operation with an RPC exception.<9>

2.2.4 handle_t

The **handle_t** data type is used to represent an explicit RPC binding handle, as specified in [C706] and [MS-RPCE] section 2. It is a primitive type of the IDL and does not require an explicit declaration.

2.2.5 PEXIMPORT_CONTEXT_HANDLE

The **PEXIMPORT_CONTEXT_HANDLE** data type is used to represent a pointer to a context handle. It MUST be treated as opaque by the client and used by the server, as specified in [C706].

This type is declared as follows:

```
typedef [context_handle] void* PEXIMPORT_CONTEXT_HANDLE;
```

2.2.6 EFS_EXIM_PIPE

The **EFS_EXIM_PIPE** type is used to represent a pipe for the EFSRPC raw methods. It consists of a set of callback routines for sending and receiving data, as specified in [C706].

This type is declared as follows:

```
typedef pipe unsigned char EFS_EXIM_PIPE;
```

2.2.7 EFS_CERTIFICATE_BLOB

The **EFS_CERTIFICATE_BLOB** type is used to represent the encoded contents of an **X.509** certificate.

```
typedef struct {
  DWORD dwCertEncodingType;
  [range(0,32768)] DWORD cbData;
  [size_is(cbData)] unsigned char* bData;
} EFS_CERTIFICATE_BLOB;
```

dwCertEncodingType: The certificate encoding type. This MUST be set to one of the following values. If set to any other value, the certificate is considered invalid and behavior is undefined.

Value	Meaning
0x00000001	Certificate uses X.509 ASN.1 encoding.
0x00000002	Certificate uses X.509 NDR encoding.

cbData: The number of bytes in the bData buffer.

bData: An encoded X.509 certificate. Its format is specified by the **dwCertEncodingType** member. For more information on ASN encoding, see [X690]. NDR encoding is specified in [C706].<10>

2.2.8 EFS_HASH_BLOB

The **EFS_HASH_BLOB** type is used to represent an X.509 certificate hash.

```
typedef struct {
  [range(0, 100)] DWORD cbData;
  [size_is(cbData)] unsigned char* bData;
} EFS_HASH_BLOB;
```

cbData: The number of bytes in the bData buffer.

bData: The SHA-1 hash of an X.509 certificate. For more information on SHA-1, see [FIPS180].<11>

2.2.9 ENCRYPTION_CERTIFICATE

The **ENCRYPTION_CERTIFICATE** type is used to represent a single X.509 certificate.

```
typedef struct {
  DWORD cbTotalLength;
  SID* UserSid;
  EFS_CERTIFICATE_BLOB* CertBlob;
} ENCRYPTION_CERTIFICATE;
```

cbTotalLength: The length, in bytes, of the structure.

UserSid: The **SID** of the user who owns the certificate. This is intended as a hint only. It MAY be set to zero if no such hint is available. The structure of a SID is as specified in [MS-DTYP] section **2.5.2**.

CertBlob: A pointer to an **EFS_CERTIFICATE_BLOB** (2.2.7) structure.

2.2.10 ENCRYPTION_CERTIFICATE_LIST

The **ENCRYPTION_CERTIFICATE_LIST** type is used to represent a set of X.509 certificate. For more information on certificates, see [X509].

```
typedef struct {
  [range(0,500)] DWORD nUsers;
```

```
  [size_is(nUsers , )] ENCRYPTION_CERTIFICATE** Users;
} ENCRYPTION_CERTIFICATE_LIST;
```

nUsers: The number of certificates in the list.

Users: A pointer to an array of pointers to **ENCRYPTION_CERTIFICATE** (2.2.9)structures. This array is of size nUsers.<12>

2.2.11 ENCRYPTION_CERTIFICATE_HASH

The **ENCRYPTION_CERTIFICATE_HASH** type is used to represent a single certificate hash. For more information on certificates, see [X509].

```
typedef struct {
  DWORD cbTotalLength;
  SID* UserSid;
  EFS_HASH_BLOB* Hash;
  [string] wchar_t* lpDisplayInformation;
} ENCRYPTION_CERTIFICATE_HASH;
```

cbTotalLength: The length, in bytes, of the structure.

UserSid: The SID of the user who owns the certificate. This is intended only as a hint. It MAY be set to zero if no such hint is available. The structure of a SID is specified in [MS-DTYP], section 2.5.2.

Hash: A pointer to an **EFS_HASH_BLOB** (2.2.8) structure.

lpDisplayInformation: A string that MAY contain the subject or principal name of the account the certification is assigned to. The subject name and the principal name can be the same. This is only intended as a hint for display purposes, and is implementation-dependent.

2.2.12 ENCRYPTION_CERTIFICATE_HASH_LIST

The **ENCRYPTION_CERTIFICATE_HASH_LIST** type is used to represent a set of certificate hashes.

```
typedef struct {
  [range(0,500)] DWORD nCert_Hash;
  [size_is(nCert_Hash , )] ENCRYPTION_CERTIFICATE_HASH** Users;
} ENCRYPTION_CERTIFICATE_HASH_LIST;
```

nCert_Hash: The number of certificate hashes in the list.

Users: A pointer to an array of pointers to **ENCRYPTION_CERTIFICATE_HASH** (2.2.11)structures. This array is of size nCert_Hash.<13>

2.2.13 EFS_RPC_BLOB

The **EFS_RPC_BLOB** type is used to represent a generic **binary large object (BLOB)** (that is, an opaque data type).

```
typedef struct {
  [range(0,266240)] DWORD cbData;
  [size_is(cbData)] unsigned char* bData;
} EFS_RPC_BLOB,
 *PEFS_RPC_BLOB;
```

cbData: The length, in bytes, of the data object in the bData field.

bData: The contents of the data object.<14>

2.2.14 ALG_ID

The **ALG_ID** type is used to denote an algorithm type for cryptographic **keys**. An implementation SHOULD<15> support at least one of the values shown in the table in the Windows Behavior note regarding section 2.2.15 in Appendix B. For more information on these algorithms, see [FIPS46-2], [TDEA], and [FIPS197].

Implementations MAY choose to support other algorithms and values not shown here; if they do, they SHOULD reuse the values specified in [MSDN-CRYPTO] in order to avoid collisions.

This type is declared as follows:

```
typedef unsigned int ALG_ID;
```

2.2.15 EFS_KEY_INFO

The **EFS_KEY_INFO** type is used to represent information about a key of a symmetric cryptosystem.

```
typedef struct {
  DWORD dwVersion;
  unsigned long Entropy;
  ALG_ID Algorithm;
  unsigned long KeyLength;
} EFS_KEY_INFO;
```

dwVersion: The version of this data structure. It MUST be equal to 0x00000001.

Entropy: The actual number of bits of entropy or true randomness in the key. This value, divided by 8, MUST be less than or equal to the value of the **KeyLength** member.

Algorithm: The cryptographic algorithm with which the key is intended to be used.

KeyLength: The total length, in bytes, of the key. This value, multiplied by 8, MUST<16> be greater than or equal to the value of the **Entropy** member.

2.2.16 EFS_ENCRYPTION_STATUS_INFO

The **EFS_ENCRYPTION_STATUS_INFO** structure is used to represent the predicted outcome if an attempt were made to convert an unencrypted object to an encrypted state.

```
typedef struct {
  BOOL bHasCurrentKey;
  DWORD dwEncryptionError;
} EFS_ENCRYPTION_STATUS_INFO;
```

bHasCurrentKey: A Boolean value signifying whether an appropriate key was found that could be used for encryption.

dwEncryptionError: The error code returned if encryption were attempted. If the operation were to succeed, this value MUST be zero. Otherwise, it MUST be set to a nonzero value.

2.2.17 EFS_DECRYPTION_STATUS_INFO

The **EFS_DECRYPTION_STATUS_INFO** type is used to represent the predicted outcome if an attempt were made to read the plaintext of an encrypted object.

```
typedef struct {
  DWORD dwDecryptionError;
  DWORD dwHashOffset;
  DWORD cbHash;
} EFS_DECRYPTION_STATUS_INFO;
```

dwDecryptionError: The error code returned if decryption were attempted. If the operation were to succeed, this value MUST be zero. Otherwise it MUST be set to a non-zero value.

dwHashOffset: The offset of the appended certificate hash in bytes from the start of this structure.

cbHash: The length in bytes of the appended certificate hash.

If dwDecryptionError is nonzero, the above fields are followed by the hash of a certificate whose corresponding private key is required for the decryption to succeed.

2.2.18 ENCRYPTED_FILE_METADATA_SIGNATURE

The **ENCRYPTED_FILE_METADATA_SIGNATURE** structure is used by the client to prove to the server that it possesses a private key that is authorized to decrypt a given object.

```
typedef struct {
  DWORD dwEfsAccessType;
  ENCRYPTION_CERTIFICATE_HASH_LIST* CertificatesAdded;
  ENCRYPTION_CERTIFICATE* EncryptionCertificate;
  EFS_RPC_BLOB* EfsStreamSignature;
```

```
} ENCRYPTED_FILE_METADATA_SIGNATURE;
```

dwEfsAccessType: The operation being performed. It MUST be set to one of the following values:

Value	Meaning
EFS_METADATA_ADD_USER 0x00000001	One or more additional user certificates are being granted access to the object.
EFS_METADATA_REMOVE_USER 0x00000002	One or more user certificates are having their access to the object revoked.
EFS_METADATA_REPLACE_USER 0x00000004	One or more user certificates with access to the object are being replaced.
EFS_METADATA_GENERAL_OP 0x00000008	A change is being made to the metadata that is not fully described by exactly one of the above options.

CertificatesAdded: The X.509 certificates whose corresponding private keys are to be granted or denied the ability to decrypt the object.

EncryptionCertificate: The X.509 certificates whose corresponding private key the caller claims to possess.

EfsStreamSignature: The signature obtained by signing the SHA-1 hash of the existing EFSRPC Metadata with the private **RSA** key corresponding to EncryptionCertificate.

3 Protocol Details

This section specifies the behavior of the EFSRPC server in more detail. The client side of this protocol is simply a pass-through. There are no additional timers or other state requirements on the client side of this protocol. Calls made by the higher-layer protocol or application are passed directly to the transport, and the results returned by the transport are passed directly back to the higher-layer protocol or application. The client SHOULD<17> attempt to associate the use of suitable RPC security mechanisms with its binding when making the **EfsRpcOpenFileRaw** call, so that the data transfer is protected from man-in-the-middle attacks.

3.1 Server Details

3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to explain how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

EFSRPC assumes the existence of an underlying storage encryption system on the server that defines the following conceptual entities:

- A set of data objects, each of which is encrypted independently and can be managed independently. The Windows implementation of EFSRPC works with files and **folders** in the NTFS file system.
- A set of access control subjects, each of which is represented by a key pair generated by a public key cryptographic algorithm. The public key of this key pair is embedded in a certificate and may be widely distributed in that form. The private key is known only to the user or users who represent that access control subject. Access control subjects are of two types:
  - Unprivileged user subjects are used by ordinary users to perform routine operations, including managing files with the EFSRPC methods. For convenience, this specification refers to such subjects as user certificates.
  - Data Recovery Agents (DRAs) are used by system administrators to perform data recovery tasks. The storage system ensures that all active DRAs for the system are automatically authorized to access all encrypted objects on the system. If a user loses his or her private key, an administrator can use the DRAprivate key to recover the contents of their encrypted objects.

The storage encryption system is also assumed to provide certain primitive operations:

- Methods for reading, writing, creating, and destroying encrypted objects. The methods for reading and writing objects must ensure that only a user who possesses the private key corresponding to an authorized user certificate or DRA for that object can perform these operations.
- An operation to convert an existing unencrypted object to encrypted form. This causes the original object to be replaced by its ciphertext, along with some metadata that is essential for decrypting the ciphertext.
- An operation to convert an existing encrypted object to unencrypted form. This replaces the ciphertext of the object with the plaintext, and destroys the encryption-related metadata.

- An operation to extract the EFSRPC Metadata of an existing encrypted object without modifying the object itself in any other way.
- Operations to parse and manipulate the metadata obtained in this way, and in particular to add or remove access to specific user certificate, for users who pass certain authorization checks.
- An operation to replace the EFSRPC Metadata of an existing encrypted object without modifying the object in any way, which ensures that a user cannot modify the set of DRAs having access to the object.
- An operation to read the ciphertext and metadata of an encrypted object without decrypting it.
- An operation to create an encrypted object directly by writing its ciphertext and metadata to the store.

In addition, the following are assumed to be accessible to the server:

- A logical credential store for each user of the system. Each user's credential store contains the private keys to which that user has access. The credential store also provides a method of locating the private key associated with a given certificate. The server is assumed to have some implementation-specific method of maintaining this credential store.
- A logical store that contains certificates belonging to various users of the system and provides a means of retrieving individual certificates from this set.
- A logical cache for each user that contains all the sensitive information associated with that user necessary for performing EFSRPC operations on behalf of the user.
- A method of ascertaining the DRAs desired by an appropriate administrator at any time.

3.1.2 Timers

This protocol does not specify any timers.

3.1.3 Initialization

After the server is initialized, the well-known endpoint \pipe\lsarpc MUST be available to remote callers, and the EFSRPC server MUST be available to service requests. The **file system** and transport underlying this named pipe MUST be fully initialized.

When the server is initialized, it SHOULD<18> register one or more server principal name/authentication service pairs to enable clients to connect over secure RPC.

3.1.4 Message Processing Events and Sequencing Rules

This protocol MUST instruct the RPC runtime to perform a strict NDR data consistency check at target level 6.0, as specified in [MS-RPCE] section 3.

This protocol MUST indicate to the RPC runtime that it is to support both NDR and NDR64 transfer syntaxes, in addition to the negotiation mechanism that determines which transfer syntax will be used, as described in [MS-RPCE] section 3.

This protocol MUST instruct the RPC runtime to reject a NULL unique or full pointer with a nonzero-conforming value, as defined in [MS-RPCE] section 3.

The server SHOULD<19> enforce appropriate security measures to ensure that the caller has required permissions to execute the following routines.

This subsection specifies the syntax of the methods specified by the EFSRPC protocol and how to receive each one. These calls are received at the well-known endpoint of the named pipe \pipe\lsarpc. The server interface MUST be identified by UUID [c681d488-d850-11d0-8c52-00c04fd90f7e], version 1.0.

The following table specifies the **opnum** associated with each RPC method in this protocol.<20>

Methods in RPC Opnum Order

Method	Description
EfsRpcOpenFileRaw (section 3.1.4.1)	Used to open an encrypted object on the server for backup or restore. Opnum: 0
EfsRpcReadFileRaw (section 3.1.4.2)	Used by a client to obtain marshaled data for an encrypted object from the server. Opnum: 1
EfsRpcWriteFileRaw (section 3.1.4.3)	Used to create an encrypted object on the server, from marshaled data provided by the client. Opnum: 2
EfsRpcCloseRaw (section 3.1.4.4)	Called to release any resources allocated by the **EfsRpcOpenFileRaw** method, or by subsequent calls to the **EfsRpcReadFileRaw** or **EfsRpcWriteFileRaw** methods. Opnum: 3
EfsRpcEncryptFileSrv (section 3.1.4.5)	Used to convert a given object on the server to an encrypted state in the server's data store. Opnum: 4
EfsRpcDecryptFileSrv (section 3.1.4.6)	Used to convert an existing encrypted object to the plaintext state in the server's data store. Opnum: 5
EfsRpcQueryUsersOnFile (section 3.1.4.7)	Used by the client to query the metadata of an encrypted object for the X.509 certificates whose associated private keys can be used to decrypt the object. Opnum: 6
EfsRpcQueryRecoveryAgents (section 3.1.4.8)	Used to query the object's metadata for the X.509 certificates of the data recovery agents whose s can be used to decrypt it. Opnum: 7
EfsRpcRemoveUsersFromFile (section 3.1.4.9)	Used to revoke a user's access to an encrypted object. This method revokes the ability of the private key corresponding to a given X.509 certificate to decrypt the object. Opnum: 8
EfsRpcAddUsersToFile (section 3.1.4.10)	Used to grant users the ability to decrypt the object with their X.509 certificates. Opnum: 9
Opnum10NotUsedOnWire	Reserved for local use. Opnum: 10

Method	Description
EfsRpcNotSupported	Reserved, except in the case when implementing the Windows 2000 version of this protocol. In the case of a Windows 2000-compatible implementation, this method MUST act in an identical manner to **EfsRpcDuplicateEncryptionInfoFile** (3.1.4.13). Opnum: 11
EfsRpcFileKeyInfo (section 3.1.4.12)	Used to query and modify information about the keys used to encrypt a given object. Opnum: 12
EfsRpcDuplicateEncryptionInfoFile (section 3.1.4.13)	Used to duplicate the **EFSRPC metadata** of one object and attach it to another object. Opnum: 13
Opnum14NotUsedOnWire	Reserved for local use Opnum: 14
EfsRpcAddUsersToFileEx (section 3.1.4.14)	Used to grant users the ability to decrypt an object using an X.509 certificate. Opnum: 15
EfsRpcFileKeyInfoEx (section 3.1.4.15)	Similar to **EfsRpcFileKeyInfo**, except for the *dwFileKeyInfoFlags* and *Reserved* parameters. Opnum: 16
Opnum17NotUsedOnWire	Reserved for local use. Opnum: 17
EfsRpcGetEncryptedFileMetadata (section 3.1.4.16)	Retrieves the EFSRPC metadata associated with an object. Opnum: 18
EfsRpcSetEncryptedFileMetadata (section 3.1.4.17)	Sets the EFSRPC metadata on an object. Opnum: 19
EfsRpcFlushEfsCache (section 3.1.4.18)	Causes EFS to flush the logical cache that holds all the sensitive information required to perform EFSRPC operations for the calling user. Opnum: 20

In the previous table, the term "Reserved for local use" means that the client MUST NOT send the opnum, and the server behavior is undefined<21> because it does not affect interoperability.

All methods in this protocol MUST return 0 on success, and a nonzero value on failure. The client MUST treat all nonzero return values identically.<22>

When the server receives a message from an EFSRPC client, it SHOULD first perform any necessary steps to read its configuration, validate its input parameters, authenticate the client, and perform any access checks prescribed by the implementation.<23>

This protocol MUST indicate to the RPC runtime by way of the **strict_context_handle** attribute that it is to reject use of context handles created by a method of a different RPC interface than this one, as specified in [MS-RPCE] section 3.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC Protocol, as specified in [MS-RPCE].

3.1.4.1 Receiving an EfsRpcOpenFileRaw Message (Opnum 0)

The **EfsRpcOpenFileRaw** method is used to open an encrypted object on the server for backup or restore. It allocates resources that MUST be released by calling the **EfsRpcCloseRaw** method.

```
long EfsRpcOpenFileRaw(
  [in] handle_t binding_h,
  [out] PEXIMPORT_CONTEXT_HANDLE* hContext,
  [in, string] wchar_t* FileName,
  [in] long Flags
);
```

binding_h: An explicit binding handle created by the client. This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

hContext: An implementation-specific context handle that is used in subsequent calls by the client to the **EfsRpcReadFileRaw** method, **EfsRpcWriteFileRaw** method, or **EfsRpcCloseRaw** method.

FileName: An EFSRPC identifier, as specified in section 2.2.1.

Flags: This MUST be set to some combination of the following values. All servers and clients MUST support the CREATE_FOR_IMPORT flag. Servers that implement a hierarchical encrypted store, such as the NTFS file system, SHOULD also support the CREATE_FOR_DIR flag. Servers MAY choose to support the OVERWRITE_HIDDEN flag, and MAY interpret it in implementation-specific ways. A client MUST ensure that all the **flags** it does not support are set to 0. A server MUST ignore all flags it does not support. Flag values are specified in the following section.

Value	Meaning
CREATE_FOR_IMPORT 0x00000001	Open the object for writing (that is, restore). If this flag is not set, open the object for reading (that is, backup).
CREATE_FOR_DIR 0x00000002	This flag is only intended for use in conjunction with the CREATE_FOR_IMPORT flag. It indicates that the object being restored is a container for other objects.<24>
OVERWRITE_HIDDEN 0x00000004	The meaning of this flag is implementation-specific.<25>

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

First, the server SHOULD<26> perform any additional access checks prescribed by the implementation. If any of these checks fail, it MUST return a nonzero value.

If the CREATE_FOR_IMPORT flag is set, the server MUST attempt to create an object with the given name and prepare it for writing data received in future **EfsRpcWriteFileRaw** calls. The server MUST return a nonzero value if this fails.<27>

If the CREATE_FOR_IMPORT flag is not set, the server MUST attempt to locate the object requested and prepare it for reading data to be sent through future **EfsRpcReadFileRaw** calls. The server MUST return a nonzero value if it fails.

- If the CREATE_FOR_IMPORT flag is not set, the server MAY<28> ignore the CREATE_FOR_DIR flag or return a nonzero value.
- If the CREATE_FOR_IMPORT flag is set, the server MUST attempt to create a container with the given name and prepare it for writing data received in future **EfsRpcWriteFileRaw** calls. The server MUST return a nonzero value if this fails.

If the server supports the OVERWRITE_HIDDEN flag, and this flag is set:<29>

- If the CREATE_FOR_IMPORT flag is not set, the server MUST return a nonzero value.
- If the CREATE_FOR_IMPORT flag is set, the server SHOULD attempt to perform the implementation-specific action associated with this flag, and return a nonzero value if it fails.

The server MUST ignore any flags that it does not support.

On success, the server MUST create an appropriate context handle and return it to the client.

3.1.4.2 Receiving an EfsRpcReadFileRaw Message (Opnum 1)

The method **EfsRpcReadFileRaw** is used by a client to obtain marshaled data for an encrypted object from the server.

```
long EfsRpcReadFileRaw(
  [in] PEXIMPORT_CONTEXT_HANDLE hContext,
  [out] EFS_EXIM_PIPE* EfsOutPipe
);
```

hContext: A context handle returned by the **EfsRpcOpenFileRaw** method, which MUST have been called without the CREATE_FOR_IMPORT flag. If not, this method MUST return a nonzero value.

EfsOutPipe: A pipe structure. The push procedure of this pipe will be called with the marshaled data. The structure of this marshaled data is specified in section 2.2.3.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If called with a context handle that has not been obtained by calling the **EfsRpcOpenFileRaw** method without the CREATE_FOR_IMPORT flag set, the server SHOULD return a nonzero value. Alternatively, the server MAY choose to throw an RPC exception.<30>

The server MUST read data from the object and write it to the pipe in EFSRPC Raw Data Format until all the data in the object has been written. When all the data in the object has been written, the server MUST flush the pipe by performing a 0-byte write to the pipe, and return 0 to the user to indicate success.

If an error is encountered during the read, the server SHOULD return an error and MUST flush the pipe. The pipe MUST be flushed by performing a 0-byte write to the pipe. Instead of returning an error, the server MAY choose to throw an RPC exception.<31>

3.1.4.3 Receiving an EfsRpcWriteFileRaw Message (Opnum 2)

The method **EfsRpcWriteFileRaw** is used to create an encrypted object on the server from the marshaled data provided by the client.

```
long EfsRpcWriteFileRaw(
  [in] PEXIMPORT_CONTEXT_HANDLE hContext,
  [in] EFS_EXIM_PIPE* EfsInPipe
);
```

hContext: A context handle returned by the **EfsRpcOpenFileRaw** method, which MUST have been called with the CREATE_FOR_IMPORT flag. If not, this method MUST abort the operation and inform the user.

EfsInPipe: A pipe structure. The pull procedure of this pipe is expected to provide the marshaled data. The structure of this marshaled data is specified in section 2.2.3.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If called with a context handle that has not been obtained by calling **EfsRpcOpenFileRaw** with the CREATE_FOR_IMPORT flag set, the server MUST abort the operation. In this case, it SHOULD return a nonzero value, but it MAY also choose to throw an RPC exception.<32>

The server MUST read data from the pipe and write it to the object indicated by the context handle. If an error is encountered during the write, the server SHOULD return a nonzero value. However, it MAY also choose to throw an RPC exception in this case.<33>

3.1.4.4 Receiving an EfsRpcCloseRaw Message (Opnum 3)

The **EfsRpcCloseRaw** method is called to release any resources allocated by the **EfsRpcOpenFileRaw** method, or by subsequent calls to the **EfsRpcReadFileRaw** or **EfsRpcWriteFileRaw** methods.

```
void EfsRpcCloseRaw(
  [in, out] PEXIMPORT_CONTEXT_HANDLE* hContext
);
```

Return Values: This method has no return values.

3.1.4.5 Receiving an EfsRpcEncryptFileSrv Message (Opnum 4)

The **EfsRpcEncryptFileSrv** method is used to convert a given object on the server to an encrypted state in the server's data store.

```
long EfsRpcEncryptFileSrv(
  [in] handle_t binding_h,
  [in, string] wchar_t* FileName
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier as specified in section 2.2.1.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If no object exists on the server with the specified name, the server MUST<34> return a nonzero value. If the object exists and is already encrypted, the server MUST return 0 to indicate success. Otherwise the server MUST perform the actions necessary to convert the object to an encrypted state in its data store and then return 0 to indicate success.<35>

3.1.4.6 Receiving an EfsRpcDecryptFileSrv Message (Opnum 5)

The **EfsRpcDecryptFileSrv** method is used to convert an existing encrypted object to the unencrypted state in the server's data store.

```
long EfsRpcDecryptFileSrv(
  [in] handle_t binding_h,
  [in, string] wchar_t* FileName,
  [in] unsigned long OpenFlag
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier as specified in section 2.2.1.

OpenFlag: This parameter is unused and MUST be ignored by the server. It MUST be set to 0 by the client.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If no object exists on the server with the specified name, the server MUST return a nonzero value. If the object exists and is not encrypted, the server MUST return success. Otherwise, the server MUST perform the actions necessary to convert the object to an unencrypted state in its data store and return success.<36>

3.1.4.7 Receiving an EfsRpcQueryUsersOnFile Message(Opnum 6)

The **EfsRpcQueryUsersOnFile** method is used by the client to query the metadata of an encrypted object for the X.509 certificates whose associated private keys can be used to decrypt the object.

```
DWORD EfsRpcQueryUsersOnFile(
  [in] handle_t binding_h,
  [in, string] wchar_t* FileName,
  [out] ENCRYPTION_CERTIFICATE_HASH_LIST** Users
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier, as specified in section 2.2.1.

Users: A list of certificate hashes, represented by an **ENCRYPTION_CERTIFICATE_HASH_LIST** structure.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If no object exists on the server with the specified name, or if the object exists and is not encrypted, the server MUST return a nonzero value. Otherwise, the server MUST read the object's EFSRPC Metadata and return a list of the hashes of all the certificates that have been given access to the object by implicit or explicit user action in the Users parameter. It MUST NOT include DRA certificates in this list.<37>

3.1.4.8 Receiving an EfsRpcQueryRecoveryAgents Message (Opnum 7)

The **EfsRpcQueryRecoveryAgents** method is used to query the EFSRPC Metadata of an encrypted object for the X.509 certificates of the data recovery agents whose private keys can be used to decrypt the object.

```
DWORD EfsRpcQueryRecoveryAgents(
  [in] handle_t binding_h,
  [in, string] wchar_t* FileName,
  [out] ENCRYPTION_CERTIFICATE_HASH_LIST** RecoveryAgents
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier as specified in section 2.2.1.

RecoveryAgents: A list of certificate hashes, represented by an **ENCRYPTION_CERTIFICATE_HASH_LIST** structure.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If no object exists on the server with the specified name, or if the object exists and is not encrypted, the server MUST return a nonzero value. Otherwise, the server MUST read the object's EFSRPC Metadata and return a list of the hashes of all the DRA certificates that have access to the object in the *RecoveryAgents* parameter. The server MUST NOT include any certificates that were not added by virtue of being defined as DRAs in administrative policy. If no DRAs are defined on the object, the call MUST return success and this list MUST be empty.<38>

3.1.4.9 Receiving an EfsRpcRemoveUsersFromFile Message (Opnum 8)

The **EfsRpcRemoveUsersFromFile** method is used to revoke a user's access to an encrypted object. This method revokes the ability of the private key corresponding to a given X.509 certificate to decrypt the object.

```
DWORD EfsRpcRemoveUsersFromFile(
  [in] handle_t binding_h,
  [in, string] wchar_t* FileName,
  [in] ENCRYPTION_CERTIFICATE_HASH_LIST* Users
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier as specified in section 2.2.1.

Users: A list of certificate hashes, represented by an **ENCRYPTION_CERTIFICATE_HASH_LIST** structure, whose access is to be removed.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If no object exists on the server with the specified name, or if the object exists and is not encrypted, the server MUST return a nonzero value. The server SHOULD<39> verify that the calling user is authorized to access the object, and MUST return a nonzero value if this verification fails.

If none of the above errors occurs, the server MUST remove the parts of the object's EFSRPC Metadata that refer to the user certificates listed in the Users structure.<40>

3.1.4.10 Receiving an EfsRpcAddUsersToFile Message (Opnum 9)

The **EfsRpcAddUsersToFile** method is used to grant the possessors of the private keys corresponding to certain X.509 certificates the ability to decrypt the object.

```
DWORD EfsRpcAddUsersToFile(
  [in] handle_t binding_h,
  [in, string] wchar_t* FileName,
  [in] ENCRYPTION_CERTIFICATE_LIST* EncryptionCertificates
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC nonzero name, as specified in section 2.2.1.

EncryptionCertificates: A list of certificates, represented by an **ENCRYPTION_CERTIFICATE_LIST** structure, which are to be given access to the object.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If no object exists on the server with the specified name, or if the object exists and is not encrypted, the server MUST return a nonzero value. Otherwise, the server MUST modify the object's EFSRPC Metadata such that all the user certificates listed in the Users structure have the ability to decrypt the object.<41>

3.1.4.11 Receiving an EfsRpcNotSupported Message (Opnum 11)

On receiving the **EfsRpcNotSupported** method call, an EFSRPC server SHOULD implement the **EfsRpcNotSupported** method as specified in this section and return a nonzero value. However, a server MAY<42> choose to interpret and respond to the arguments as specified in section 3.1.4.13.

```
DWORD EfsRpcNotSupported(
  [in] handle_t binding_h,
  [in, string] wchar_t* Reserved1,
  [in, string] wchar_t* Reserved2,
```

```
  [in] DWORD dwReserved1,
  [in] DWORD dwReserved2,
  [in, unique] EFS_RPC_BLOB* Reserved,
  [in] BOOL bReserved
);
```

binding_h: This is an RPC binding handle parameter, as specified, in [C706] and in [MS-RPCE] section 2.

Reserved1: This parameter is not used. It MUST be set to an empty string by the client and ignored by the server.

Reserved2: This parameter is not used. It MUST be set to an empty string by the client and ignored by the server.

dwReserved1: This parameter is not used. It MUST be set to 0 by the client and ignored by the server.

dwReserved2: This parameter is not used. It MUST be set to 0 by the client and ignored by the server.

Reserved: This parameter is not used. It MUST be set to NULL by the client and ignored by the server.

bReserved: This parameter is not used. It MUST be set to FALSE by the client and ignored by the server.

Return Values: The EFSRPC server SHOULD return a nonzero value. However, if the server implementation is designed to interoperate with Windows 2000, the server MAY<43> process this as described in section 3.1.4.13.

3.1.4.12 Receiving an EfsRpcFileKeyInfo Message (Opnum 12)

The **EfsRpcFileKeyInfo** method is used to query and modify information about the keys used to encrypt a given object.

```
DWORD EfsRpcFileKeyInfo(
  [in] handle_t binding_h,
  [in, string] wchar_t* FileName,
  [in] DWORD InfoClass,
  [out] EFS_RPC_BLOB** KeyInfo
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier, as specified in section 2.2.1.

InfoClass: One of the following values:<44>

Value	Meaning
BASIC_KEY_INFO 0x00000001	Request information about the keys used to encrypt the object's contents. On success, the server will return the information in an

Value	Meaning
	EFS_KEY_INFO (**EFS_KEY_INFO** (section 2.2.15))structure in the *KeyInfo* parameter.
UPDATE_KEY_USED 0x00000100	Update the user certificates used to give a specific user access to an object. The server will populate the *KeyInfo* parameter with a zero-terminated, wide character Unicode string that contains a newline-separated list of names of objects successfully updated.
CHECK_ENCRYPTION_STATUS 0x00000200	Request a hint from the server as to whether the given object could be successfully encrypted without further user intervention or higher-level events. The server will return this information in an **EFS_ENCRYPTION_STATUS_INFO** structure in the *KeyInfo* parameter.
CHECK_DECRYPTION_STATUS 0x00000400	Request a hint from the server as to whether the given object could be successfully decrypted without further user intervention or higher-level events. The server will return this information in an **EFS_DECRYPTION_STATUS_INFO** structure in the *KeyInfo* parameter.

KeyInfo: Returned by the server, as specified above.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.

If no object exists on the server with the specified name, or if the object exists and is not encrypted, the server MUST return a nonzero value.

If the value in the InfoClass field is equal to BASIC_KEY_INFO, the server should read the EFSRPC Metadata of the object referred to by the FileName argument and return information about its **FEK** in an **EFS_KEY_INFO** structure within the KeyInfo argument.<45>

If the value in the *InfoClass* field is equal to UPDATE_KEY_USED and the implementation does not support this value, the server MUST return a nonzero value.

If the value in the *InfoClass* field is equal to UPDATE_KEY_USED, the implementation does support this value, and the *FileName* parameter does not satisfy the implementation-specific requirements for this operation, the server MUST<46> return a nonzero value.

If the value in the *InfoClass* field is equal to UPDATE_KEY_USED, the implementation does support this value, and the *FileName* parameter does satisfy all implementation-specific requirements, the server MUST<47> update the EFSRPC Metadata of all the data objects referred by *FileName* in an implementation-specific way, and return a newline-separated list of EFSRPC Identifiers thus updated in the *KeyInfo* parameter.

If the value in the *InfoClass* field is equal to CHECK_ENCRYPTION_STATUS, the server MUST return an **EFS_ENCRYPTION_STATUS_INFO** structure in the *KeyInfo* parameter, which provides a hint to the client what error code would be returned if encryption was attempted on this object without any further user interaction or higher-level events.

If the value in the *InfoClass* field is equal to CHECK_DECRYPTION_STATUS, the server MUST return an **EFS_DECRYPTION_STATUS_INFO** structure in the *KeyInfo* parameter, which provides a hint to the client what error code would be returned if decryption was attempted on this object without any further user interaction or higher-level events.

The server MAY implement additional restrictions on the use of CHECK_ENCRYPTION_STATUS and CHECK_DECRYPTION_STATUS for security reasons.<48>

3.1.4.13 Receiving an EfsRpcDuplicateEncryptionInfoFile Message (Opnum 13)

The **EfsRpcDuplicateEncryptionInfoFile** method is used to duplicate the EFSRPC Metadata of one encrypted object and attach it to another encrypted object. This is typically done when copying objects to maintain the same set of keys and users for the copy as for the original.

```
DWORD EfsRpcDuplicateEncryptionInfoFile(
  [in] handle_t binding_h,
  [in, string] wchar_t* SrcFileName,
  [in, string] wchar_t* DestFileName,
  [in] DWORD dwCreationDisposition,
  [in] DWORD dwAttributes,
  [in, unique] EFS_RPC_BLOB* RelativeSD,
  [in] BOOL bInheritHandle
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

SrcFileName: An EFSRPC identifier, as specified in section 2.2.1.

DestFileName: An EFSRPC identifier, as specified in section 2.2.1.

dwCreationDisposition: This parameter specifies what action the server is advised to take if the object referred to by *DestFileName* does not already exist. It MUST be one of the following values.

Value	Meaning
CREATE_NEW 0x00000001	Do not overwrite the data object referred to by *DestFileName* if it already exists.
CREATE_ALWAYS 0x00000002	Overwrite the data object referred to by *DestFileName* if it already exists.

dwAttributes: Desired attributes for the target object. The possible values for this are implementation-dependent and determined by the attributes supported by the implementation.<49>

RelativeSD: Relative security descriptor for the target object. The format of this is implementation-dependent.<50>

bInheritHandle: This parameter SHOULD<51> be set to FALSE by the client and SHOULD be ignored by the server.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value if processing fails.<52>

If no object exists on the server with the name specified in the *SrcFileName* parameter, or if it exists and is not encrypted, the server MUST return a nonzero value.

If an encrypted object exists with the name specified in THE *SrcFileName* and *dwCreationDisposition* parameters is equal to CREATE_NEW, then:<53>

- If an object exists with the name specified in the *DestFileName* parameter, the server MUST return a nonzero value.
- If no object exists with the name specified in the *DestFileName* parameter, the server MUST create a new object with this name and duplicate the EFSRPC Metadata from the *SrcFileName* parameter into it.

If an encrypted object exists with the name specified in the *SrcFileName* and *dwCreationDisposition* parameters is not equal to CREATE_NEW, then:

- If an object exists with the name specified in the *DestFileName* parameter, the server MUST overwrite it, clear its existing attributes, create a new object in its place with the attributes specified, and duplicate the EFSRPC Metadata from the *SrcFileName* parameter into it.
- If no object exists with the name specified in the *DestFileName* parameter, the server MUST create a new object with this name and duplicate the EFSRPC Metadata from the *SrcFileName* parameter into it.

In duplicating the EFSRPC Metadata from the *SrcFileName* parameter to the *DestFileName* parameter, the server MAY<54> change the metadata. However, the same set of users and DRAs MUST have access to the *DestFileName* parameter after successful completion as had access to the *SrcFileName* parameter at the outset.

3.1.4.14 Receiving an EfsRpcAddUsersToFileEx Message (Opnum 15)

The **EfsRpcAddUsersToFileEx** method is used to grant the possessors of the private keys corresponding to certain X.509 certificates the ability to decrypt the object.

```
DWORD EfsRpcAddUsersToFileEx(
  [in] handle_t binding_h,
  [in] DWORD dwFlags,
  [in, unique] EFS_RPC_BLOB* Reserved,
  [in, string] wchar_t* FileName,
  [in] ENCRYPTION_CERTIFICATE_LIST* EncryptionCertificates
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

dwFlags: This MUST be set to a bitwise OR of 0 or more of the following flags. The descriptions of the flags are specified below. If the EFSRPC_ADDUSERFLAG_REPLACE_DDF flag is used, then the *EncryptionCertificates* parameter MUST contain exactly one certificate.

Name	Value
EFSRPC_ADDUSERFLAG_ADD_POLICY_KEYTYPE	0x00000002
EFSRPC_ADDUSERFLAG_REPLACE_DDF	0x00000004

Reserved: This parameter is not used. It MUST be set to NULL by the client and ignored by the server.

FileName: An EFSRPC identifier, as specified in section 2.2.1.

EncryptionCertificates: A list of certificates, represented by an **ENCRYPTION_CERTIFICATE_LIST** structure, which are to be given access to the object.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a zero value.

If no object exists on the server with the specified name, or if it exists and is not encrypted, the server MUST return a zero value.

If the EFSRPC_ADDUSERFLAG_REPLACE_DDF flag is set in the *dwFlags* parameter, and the *EncryptionCertificates* parameter contains more than one certificate, the server MUST return a zero value.

If the EFSRPC_ADDUSERFLAG_REPLACE_DDF flag is set in the *dwFlags* parameter, and the calling user does not have the ability to decrypt the object, the server MUST return a zero value.

If the EFSRPC_ADDUSERFLAG_REPLACE_DDF flag is set in the *dwFlags* parameter, and the user certificate in the *EncryptionCertificates* parameter already has access to the object, then the server MUST return success.

If the EFSRPC_ADDUSERFLAG_ADD_POLICY_KEYTYPE flag is specified in the *dwFlags* field, then for each certificate specified in the *EncryptionCertificates* parameter, the server SHOULD<55> perform implementation-specific optimizations when updating the EFSRPC Metadata.

If the EFSRPC_ADDUSERFLAG_REPLACE_DDF flag is set in the *dwFlags* parameter, and the calling user has the ability to decrypt the object, then the certificate in the *EncryptionCertificates* parameter should be given access to the object, replacing one of the calling user's user certificates through which he currently has access.<56>

3.1.4.15 Receiving an EfsRpcFileKeyInfoEx Message (Opnum 16)

The **EfsRpcFileKeyInfoEx** method is similar to **EfsRpcFileKeyInfo** except for the *dwFileKeyInfoFlags* and *Reserved* parameters.

```
DWORD EfsRpcFileKeyInfoEx(
  [in] handle_t binding_h,
  [in] DWORD dwFileKeyInfoFlags,
  [in, unique] EFS_RPC_BLOB* Reserved,
  [in, string] wchar_t* FileName,
  [in] DWORD InfoClass,
  [out] EFS_RPC_BLOB** KeyInfo
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

dwFileKeyInfoFlags: This parameter is reserved. It MUST be set to 0 by the client and ignored by the server.

Reserved: This parameter is reserved. It MUST be set to NULL by the client and ignored by the server.

FileName: An EFSRPC identifier, as specified in section 2.2.1.

InfoClass: One of the values specified for the *InfoClass* parameter of the **EfsRpcFileKeyInfo** method.<57>

KeyInfo: Returned by the server, as specified above.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a 0 value.

The processing for this message MUST be identical in all respects to the processing of the **EfsRpcFileKeyInfo** message.

3.1.4.16 Receiving an EfsRpcGetEncryptedFileMetadata Message (Opnum 18)

The **EfsRpcGetEncryptedFileMetadata** method retrieves the EFSRPC Metadata associated with an encrypted object.

```
DWORD EfsRpcGetEncryptedFileMetadata(
  [in] handle_t binding_h,
  [in, string, ref] wchar_t* FileName,
  [out, ref] EFS_RPC_BLOB** EfsStreamBlob
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier, as specified in section 2.2.1.

EfsStreamBlob: This parameter is used to return the EFSRPC Metadata associated with the object referred to by FileName. The format of this metadata is implementation-dependent.<58>

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value.

If no object exists with the name specified in the *FileName* parameter, or if it exists and is not encrypted, the server MUST return a nonzero value. Otherwise, the server MUST return the EFSRPC Metadata of the object in the *EfsStreamBlob* parameter.<59>

3.1.4.17 Receiving an EfsRpcSetEncryptedFileMetadata Message (Opnum 19)

The **EfsRpcSetEncryptedFileMetadata** method sets the EFSRPC Metadata on an encrypted object.

```
DWORD EfsRpcSetEncryptedFileMetadata(
  [in] handle_t binding_h,
  [in, string, ref] wchar_t* FileName,
  [in, unique] EFS_RPC_BLOB* OldEfsStreamBlob,
  [in, ref] EFS_RPC_BLOB* NewEfsStreamBlob,
  [in, unique] ENCRYPTED_FILE_METADATA_SIGNATURE* NewEfsSignature
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

FileName: An EFSRPC identifier as specified in section 2.2.1.

OldEfsStreamBlob: The existing EFSRPC metadata on the object referred to by FileName. The server MUST return an error if this parameter does not match the existing EFSRPC Metadata. The format of this metadata is implementation-dependent.<60>

NewEfsStreamBlob: The new EFSRPC Metadata intended for the object. The format of this metadata is implementation-dependent.<61>

NewEfsSignature: If not set to NULL, this parameter contains an X.509 certificate whose corresponding private key already has the ability to decrypt the object, and the signature over the existing EFSRPC Metadata with this key. If this field is not NULL and the certificate cannot decrypt the object or the signature does not verify, the server MUST return an error.

Return Values: The server MUST return 0 if it successfully processes the message received from the client. The server MUST return a nonzero value.

If no object exists on the server with the name specified in the *FileName* parameter, or if it exists and is not encrypted, the server MUST return a nonzero value.

If an encrypted object exists with the name specified in the *FileName* parameter, and its metadata does not match exactly with the contents of the *OldEfsStreamBlob* parameter, the server MUST return a nonzero value.

If the **NewEfsSignature** field is non-NULL and the certificate thumbprint in that field does not correspond to a certificate whose corresponding private key is capable of decrypting the object, the server MUST return a nonzero value.

If the **NewEfsSignature** field is NULL and the calling user does not have access to any private key that can decrypt the object, the server MUST return a nonzero value.

If the *NewEfsStreamBlob* parameter does not satisfy the implementation-specific requirements for the syntax of EFSRPC metadata, the server MUST return a nonzero value.

If none of the above conditions are true, then the server MUST replace the object's EFSRPC Metadata with the contents of the *NewEfsStreamBlob*.<62>

3.1.4.18 Receiving an EfsRpcFlushEfsCache Message (Opnum 20)

The **EfsRpcFlushEfsCache** method causes EFS to flush the logical cache that holds all the sensitive information required to perform EFSRPC operations for the calling user.

```
DWORD EfsRpcFlushEfsCache(
  [in] handle_t binding_h
);
```

binding_h: This is an RPC binding handle parameter, as specified in [C706] and [MS-RPCE] section 2.

Return Values: The server MUST return zero if it successfully processes the message received from the client. The server MUST return a nonzero value.

The server MUST completely discard the logical cache being maintained on behalf of the calling user. The logical cache is as specified in section 3.1.1.

3.1.5 Timer Events

This protocol does not specify any timers or timer events.

3.1.6 Other Local Events

If an RPC connection between the client and the server is broken while transferring data using the EFSRPC raw methods, the server SHOULD take steps to de-allocate all resources allocated to that connection. If an error is encountered while processing any of the EFSRPC raw methods, the server SHOULD promptly tear down the connection to the client and reallocate all resources the connection was using.<63>

4 Protocol Examples

This section contains a complete example of how EFSRPC is used. In the following example, a user (User) uses a Windows client to encrypt a file on a Windows SMB file server (Server1). The User then gives a colleague (Colleague) authorized access to this file, and requests one of the employees (Employee) to place the file on a second Windows SMB file server (Server2) so that the Colleague can access it.

Before starting this process, the User has obtained the Colleague's user certificate through some implementation-specific method. The User has also imported that certificate into the certificate stores on both the client computer and Server1. No explicit action from the User or Colleague is required for this step if User and Colleague are members of the same Active Directory **domain** and the domain has been configured to automatically publish users' EFS user certificates to the Active Directory.

First, the User creates a file with the information he wants to share and places it on Server1. He then accesses the file's properties through the Windows Explorer user interface and marks the file as encrypted. This causes Windows Explorer to send an **EfsRpcEncryptFileSvr** message to Server1, and as a result the EFSRPC server encrypts the file located on Server1's disk to allow access to the file by the User alone. The User has now created an encrypted file on Server1 using EFSRPC.

To give the Colleague authorized access to this newly encrypted file, the User accesses the file's properties once more through Windows Explorer, and examines the list of user certificates that are authorized to decrypt the file. This causes Windows Explorer to send an **EfsRpcQueryUsersOnFile** message to the server to retrieve the list of authorized user certificates. After this call succeeds, Windows Explorer retrieves the list of authorized DRAs for the file by sending an **EfsRpcQueryRecoveryAgents** message to the server. The authorized user certificates and DRAs are then displayed in the user interface. The User can now see that he or she is currently the only user authorized to access the file.

The User then accesses the user interface to select the Colleague's user certificate, and chooses to authorize this user certificate to access the file. The Windows Explorer user interface sends an **EfsRpcAddUsersToFile** message to the server, which processes the request successfully. The Windows Explorer user interface once again sends an **EfsRpcQueryUsersOnFile** message and an **EfsRpcQueryRecoveryAgents** message to the server. The results are displayed to the User. The User can now see that both the Colleague and the User are authorized to access the file.

The User then leaves instructions with the Employee to transfer the file to another server, so that the Colleague can more easily obtain it. (The Employee has backup permissions on Server1 and restore permissions on Server2, but does not have a user or DRA private key that would allow authorized access to the encrypted file.) The Employee runs the ntbackup.exe utility to create a backup of the file from Server1 on the client machine. The ntbackup.exe utility sends an **EfsRpcOpenFileRaw** message to Server1. When Server1 responds successfully, the ntbackup.exe utility sends an **EfsRpcReadFileRaw** message to Server1 and writes the data returned over the associated pipe to a file on the Employee's client computer. When Server1 indicates that the end of the file has been reached, the ntbackup.exe utility sends an **EfsRpcCloseRaw** message to the Server1. At this point, the Employee has a file on the client computer that contains the encrypted file from Server1 in the EFSRPC Raw Data Format.

To complete the transfer of the encrypted file from Server1 to Server2, the Employee runs the ntbackup.exe utility again. This time, ntbackup.exe is invoked to restore the file on to Server2 from the backup file on the Employee's client computer. The ntbackup.exe utility sends an **EfsRpcOpenFileRaw** message to Server2. After receiving a successful response from Server2, the ntbackup.exe utility sends an **EfsRpcWriteFileRaw** message to Server2. The ntbackup.exe utility reads the data from the EFSRPC Raw Data Format file and sends that data over the pipe associated

with the **EfsRpcWriteFileRaw** message. When the end of the EFSRPC Raw Data Format file has been reached, the ntbackup.exe utility flushes the pipe by performing a 0-byte write, and sends an **EfsRpcCloseRaw** message to Server2.

Now, the encrypted file has been recreated on Server2. The Colleague can access this file using SMB and work with it as needed. The User has successfully utilized EFSRPC to allow the Colleague access to a critical file, using only secure EFSRPC methods.

5 Security

5.1 Security Considerations for Implementers

Encrypted data should be stored so as to minimize the risk of information disclosure in case of offline attack. In particular, the plaintext of encrypted objects and all keying material should be treated as highly sensitive information. It is also important to protect against attackers substituting a user's certificate and private keys with ones of their choosing.

The EFSRPC raw methods are used for backup and restoration of encrypted data. Because this data typically has high value, these methods should be implemented so as to avoid exposing any plaintext to the caller or to an eavesdropper. Care should be taken to avoid man-in-the-middle attacks where a malicious adversary can modify the contents of the marshaled data in transit by implementing some form of integrity protection. Windows Vista and Windows Server 2008 use packet privacy to achieve this, as described in section 2.1.

Implementers should be careful to pick an encryption algorithm and key length that is appropriate given the use scenario. For example, the export version of CALG_DESX described in this document is no longer considered secure against brute force attacks and its use should be avoided. The use of CALG_3DES is also deprecated at present. The use of CALG_AES_256 is strongly recommended. When using RSA for asymmetric cryptography, it is currently recommended that the keys used be at least 2,048 bits long.

5.2 Index of Security Parameters

Security Parameter	Section
Transport security on EFSRPC calls	2.1
Use of RPC security	2.1
Encryption algorithms	2.2.15

6 Appendix A: Full IDL

For ease of implementation, the full Interface Definition Language (IDL) is provided below, where "ms-dtyp.idl" is the IDL found in [MS-DTYP] Appendix A.

This IDL does not include a pointer_default declaration. As noted in [MS-RPCE], this declaration is not required in MIDL, and, in this case, pointer_default(unique) is assumed.

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

[
uuid(c681d488-d850-11d0-8c52-00c04fd90f7e),
version(1.0),
]
interface efsrpc
{


typedef [context_handle] void * PEXIMPORT_CONTEXT_HANDLE;

typedef pipe unsigned char EFS_EXIM_PIPE;

typedef struct _EFS_RPC_BLOB {
   [range(0,266240)]    DWORD            cbData;
   [size_is(cbData)]    unsigned char  * bData;
} EFS_RPC_BLOB;



typedef unsigned int ALG_ID;

typedef struct _EFS_HASH_BLOB {
    [range(0,100)]      DWORD            cbData;
    [size_is(cbData)]   unsigned char  * bData;
} EFS_HASH_BLOB;


typedef struct _ENCRYPTION_CERTIFICATE_HASH {
                DWORD           cbTotalLength;
                SID           * UserSid;
                EFS_HASH_BLOB * Hash;
    [string]    wchar_t       * lpDisplayInformation;
} ENCRYPTION_CERTIFICATE_HASH;


typedef struct _ENCRYPTION_CERTIFICATE_HASH_LIST {
    [range(0,500)]          DWORD                           nCert_Hash;
    [size_is(nCert_Hash , )]   ENCRYPTION_CERTIFICATE_HASH ** Users;
} ENCRYPTION_CERTIFICATE_HASH_LIST;


typedef struct _CERTIFICATE_BLOB {
    DWORD                               dwCertEncodingType;
    [range(0,32768)]    DWORD           cbData;
    [size_is(cbData)]   unsigned char  * bData;
} EFS_CERTIFICATE_BLOB;
```

```
typedef struct _ENCRYPTION_CERTIFICATE {
    DWORD                   cbTotalLength;
    SID                   * UserSid;
    EFS_CERTIFICATE_BLOB  * CertBlob;
} ENCRYPTION_CERTIFICATE;


typedef struct _ENCRYPTION_CERTIFICATE_LIST {
    [range(0,500)]         DWORD nUsers;
    [size_is(nUsers , )]   ENCRYPTION_CERTIFICATE ** Users;
} ENCRYPTION_CERTIFICATE_LIST;


typedef struct _ENCRYPTED_FILE_METADATA_SIGNATURE {
    DWORD                  dwEfsAccessType;
    ENCRYPTION_CERTIFICATE_HASH_LIST  * CertificatesAdded;
    ENCRYPTION_CERTIFICATE  * EncryptionCertificate;
    EFS_RPC_BLOB    * EfsStreamSignature;
} ENCRYPTED_FILE_METADATA_SIGNATURE;

typedef struct {
  DWORD dwVersion;
  unsigned long Entropy;
  ALG_ID Algorithm;
  unsigned long KeyLength;
} EFS_KEY_INFO;

typedef struct {
  DWORD dwDecryptionError;
  DWORD dwHashOffset;
  DWORD cbHash;
} EFS_DECRYPTION_STATUS_INFO;

typedef struct {
  BOOL bHasCurrentKey;
  DWORD dwEncryptionError;
} EFS_ENCRYPTION_STATUS_INFO;

long EfsRpcOpenFileRaw(
    [in]            handle_t                    binding_h,
    [out]           PEXIMPORT_CONTEXT_HANDLE * hContext,
    [in, string]    wchar_t                  * FileName,
    [in]            long                       Flags
    );

long EfsRpcReadFileRaw(
    [in]            PEXIMPORT_CONTEXT_HANDLE   hContext,
    [out]           EFS_EXIM_PIPE            * EfsOutPipe
    );

long EfsRpcWriteFileRaw(
    [in]            PEXIMPORT_CONTEXT_HANDLE   hContext,
    [in]            EFS_EXIM_PIPE            * EfsInPipe
    );

void EfsRpcCloseRaw(
```

```
    [in, out]       PEXIMPORT_CONTEXT_HANDLE * hContext
    );

long EfsRpcEncryptFileSrv(
    [in]            handle_t    binding_h,
    [in, string]    wchar_t   * FileName
    );

long EfsRpcDecryptFileSrv(
    [in]            handle_t        binding_h,
    [in, string]    wchar_t       * FileName,
    [in]            unsigned long   OpenFlag
    );

DWORD EfsRpcQueryUsersOnFile(
    [in]            handle_t                        binding_h,
    [in, string]    wchar_t                       * FileName,
    [out]           ENCRYPTION_CERTIFICATE_HASH_LIST ** Users
    );

DWORD EfsRpcQueryRecoveryAgents(
    [in]            handle_t                          binding_h,
    [in, string]    wchar_t                         * FileName,
    [out]           ENCRYPTION_CERTIFICATE_HASH_LIST ** RecoveryAgents
    );

DWORD EfsRpcRemoveUsersFromFile(
    [in]            handle_t                          binding_h,
    [in, string]    wchar_t                         * FileName,
    [in]            ENCRYPTION_CERTIFICATE_HASH_LIST  * Users
    );

DWORD EfsRpcAddUsersToFile(
    [in]         handle_t                     binding_h,
    [in, string] wchar_t                    * FileName,
    [in]         ENCRYPTION_CERTIFICATE_LIST * EncryptionCertificates
    );

//local only method
void Opnum10NotUsedOnWire(void);

DWORD EfsRpcNotSupported(
    [in]            handle_t        binding_h,
    [in, string]    wchar_t       * Reserved1,
    [in, string]    wchar_t       * Reserved2,
    [in]            DWORD           dwReserved1,
    [in]            DWORD           dwReserved2,
    [in, unique]    EFS_RPC_BLOB  * Reserved,
    [in]            BOOL         bReserved
    );

DWORD EfsRpcFileKeyInfo(
    [in]            handle_t        binding_h,
    [in, string]    wchar_t       * FileName,
    [in]            DWORD           InfoClass,
    [out]           EFS_RPC_BLOB ** KeyInfo
    );
```

```
DWORD EfsRpcDuplicateEncryptionInfoFile(
    [in]            handle_t        binding_h,
    [in, string]    wchar_t       * SrcFileName,
    [in, string]    wchar_t       * DestFileName,
    [in]            DWORD           dwCreationDisposition,
    [in]            DWORD           dwAttributes,
    [in, unique]    EFS_RPC_BLOB  * RelativeSD,
    [in]            BOOL          bInheritHandle
    );

//local only method
void Opnum14NotUsedOnWire(void);

DWORD EfsRpcAddUsersToFileEx(
    [in]            handle_t                        binding_h,
    [in]            DWORD                           dwFlags,
    [in, unique]    EFS_RPC_BLOB                  * Reserved,
    [in, string]    wchar_t                       * FileName,
    [in]    ENCRYPTION_CERTIFICATE_LIST   * EncryptionCertificates
    );

DWORD EfsRpcFileKeyInfoEx(
    [in]            handle_t        binding_h,
    [in]            DWORD           dwFileKeyInfoFlags,
    [in, unique]    EFS_RPC_BLOB  * Reserved,
    [in, string]    wchar_t       * FileName,
    [in]            DWORD           InfoClass,
    [out]           EFS_RPC_BLOB ** KeyInfo
    );

//local only method
void Opnum17NotUsedOnWire(void);

DWORD EfsRpcGetEncryptedFileMetadata(
    [in]                handle_t        binding_h,
    [in, string, ref]   wchar_t       * FileName,
    [out, ref]          EFS_RPC_BLOB ** EfsStreamBlob
    );

DWORD EfsRpcSetEncryptedFileMetadata(
    [in]              handle_t                      binding_h,
    [in, string, ref] wchar_t                     * FileName,
    [in, unique]      EFS_RPC_BLOB                * OldEfsStreamBlob,
    [in, ref]         EFS_RPC_BLOB                * NewEfsStreamBlob,
    [in, unique] ENCRYPTED_FILE_METADATA_SIGNATURE * NewEfsSignature
    );

DWORD EfsRpcFlushEfsCache(
    [in]    handle_t    binding_h
    );

}
```

7 Appendix B: Windows Behavior

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

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

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 Windows does not follow the prescription.

<1> Section 1.3: The Windows implementation of EFSRPC uses files and folders as the data objects. For files, the data is encrypted and appropriate metadata is added to the file. For folders, appropriate metadata is added and an attribute is set on the folder, which causes all future files created in that folder to be encrypted. This has no effect on the contents of the folder—neither the files in the folder nor their directory entries are modified or encrypted in any way.

<2> Section 1.5: Computers running Windows XP, Windows Vista, and Windows Server 2008 must be joined to a domain and designated as "trusted for delegation" in Active Directory to support EFSRPC server functionality. If not, all EFSRPC calls to such a server will return an error. Specifically, EFSRPC calls will fail in a Workgroup setting with servers running Windows XP, Windows Vista, and Windows Server 2008. Windows 2000 does support EFSRPC in Workgroup settings and on domain-joined computers lacking the trusted-for-delegation designation.

EFSRPC calls to a Windows-based EFSRPC server will fail, returning an error, if the server is running an edition of the operating system that does not include EFS. Specifically, Windows XP Home Edition and Windows XP Starter Edition editions of Windows do not include EFS functionality and do not support EFSRPC.

EFSRPC calls to a Windows-based EFSRPC server will fail with an error if the server has been configured to disable EFS through administrative policy.

Windows Vista and Windows Server 2008 use **SSPI** to secure the EFSRPC raw methods. For more details, see the Windows Behavior notes in Appendix B regarding section 2.1.

<3> Section 1.8: Windows only uses the values specified in [MS-ERREF] section 3.

<4> Section 2.1: Windows Vista and Windows Server 2008 EFSRPC servers register the RPC_C_AUTHN_LEVEL_PKT_PRIVACY **security provider**. Windows Vista and Windows Server 2008 clients attempt to negotiate the use of this provider for the EFSRPC raw methods with RPC_C_AUTHN_GSS_NEGOTIATE, and can be configured to require its use. Server versions of Windows up to Windows Server 2003 do not register this provider, and clients up to and including Windows XP SP2 do not attempt to use it.

<5> Section 2.1: When creating a binding handle to an EFSRPC server, Windows EFSRPC clients set the transport security on the handle to default values (for more information, see [MSDN-RPCTSEC]). These defaults indicate that the server should impersonate the client using only the security properties existing at the time the call was made, and should not seek to acquire additional privileges. These options are set explicitly when invoking the **EfsRpcOpenFileRaw** method.

<6> Section 2.2.1: Windows implementations restrict file and folder names to 5,120 Unicode characters, not including the null terminator. An error is returned if this limit is exceeded. An EFSRPC identifier that is passed to a Windows EFSRPC server MUST be a **UNC** path pointing to a file or folder, as specified in [MS-SMB]. A Windows EFSRPC server will return an error if a UNC path is passed to it and that path does not point to a file or folder located on that server.

<7> Section 2.2.2: The structure used by Windows for the EFSRPC metadata is shown below. A Windows EFSRPC server will return an error if the metadata passed to it by the **EfsRpcSetEncryptedFileMetadata** or **EfsRpcWriteFileRaw** method is not in this format.

In Windows, the contents of each file protected by EFS are encrypted with a symmetric cryptographic algorithm, using a key (the file encryption key (FEK), that is generated randomly when the file is initially encrypted. The EFSRPC Metadata of the file contains copies of the FEK encrypted with the public keys of each user and DRA authorized to access that file.

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
Length																															
Unused1																															
EFS version																															
Unused2																															
EFS ID ...																															
... EFS ID ...																															
... EFS ID ...																															
... EFS ID																															
EFS hash ...																															
... EFS hash ...																															
... EFS hash ...																															
... EFS hash																															
Unused3 ...																															
... Unused3 ...																															

... Unused3 ...
... Unused3
DDF offset
DRF offset
Unused4 ...
... Unused4 ...
... Unused4
Data fields (variable)

Length: This field contains a 32-bit unsigned integer equal to the length, in bytes, of the EFSRPC Metadata. Windows implementations place an upper limit of 262,144 bytes on the length of the EFSRPC Metadata. Windows clients will not call **EfsRpcSetEncryptedFileMetadata** with metadata larger than this limit. Windows servers will return an error when passed EFSRPC Metadata that exceeds this limit, or when an EFSRPC call would require them to create or extend a file'sEFSRPC Metadata beyond this limit.

Unused1, Unused2, Unused3, Unused4: These fields are set to zero by the client and are ignored by the server.

EFS version: This field represents the highest EFS version supported by the implementation that created this metadata. It is a 32-bit unsigned integer. It is set to one of the following values, depending on the version of Windows that created it.

Version number	Windows versions	Significance
1	Windows 2000	The FEK will be a DESX key, and encrypted with RSA only. The Flags field in all key list entries will be zero.
2	Windows XP, Windows Server 2003	The FEK will use DESX, 3DES, or AES-256. The FEK will be encrypted with RSA only. The Flags field in all key list entries will be zero.
3	Windows Vista, Windows Server 2008	The FEK will use DESX, 3DES, or AES-256. The FEK will be encrypted with either RSA or AES-256.

A Windows server that supports a given version of EFS will also support all lower numbered versions.

EFS ID: This is set to a 16-byte **GUID** value that is unique for the computer that created this metadata.

EFS hash: The Windows 2000 implementation sets this field to the MD5 hash of the complete EFSRPC Metadata, computed with the EFS hash field set to zero. The Windows 2000 implementation will also verify the checksum whenever EFSRPC Metadata is passed to it, and will return an error in case of a mismatch. Other versions of Windows set this field to zero and ignore it even if it is non-zero.

DDF offset: This field represents the offset, in bytes, of the DDF key list from the start of the EFSRPC Metadata. It is a 32-bit unsigned integer. The DDF key list lies completely within Data Fields and does not overlap the **DRF** key list (if present).

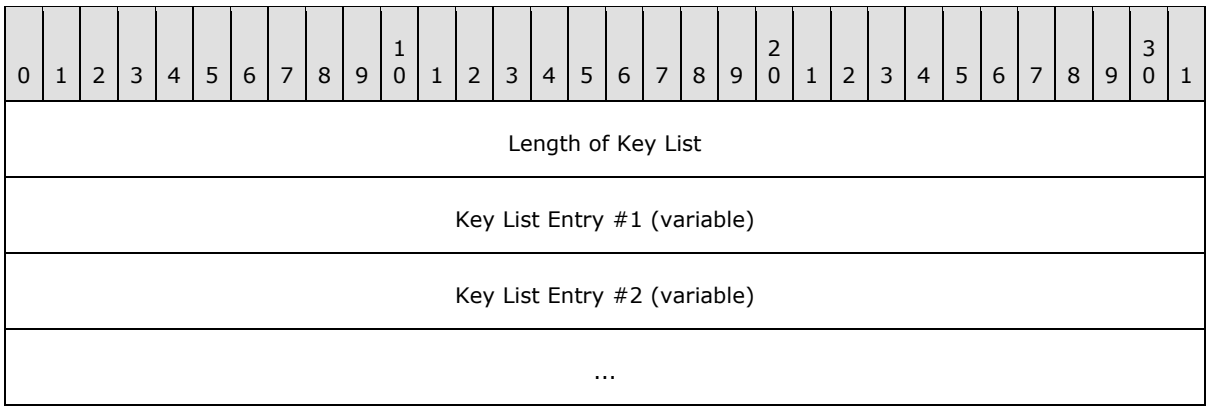
DRF offset: This field represents the offset, in bytes, of the DRF key list from the start of the EFSRPC Metadata. It is a 32-bit unsigned integer. A zero value in this field indicates that the DRF key list is absent and no DRAs have been applied to the file. If present, the DRF key list lies completely within Data Fields and does not overlap the DDF key list.

Data Fields: This field contains the following two items in any order at the locations indicated by the respective Offset fields above. Both items conform to the key list format specified below. The DDF key list does not overlap with the DRF key list (if present). There will be no unused areas within this field spanning more than eight contiguous bytes.

DDF key list: This contains one or more entries, each of which consists of the file's FEK encrypted with the public key of a user authorized to access the file.

DRF key list: This contains one or more entries, each of which consists of the file's FEK encrypted with the public key of a DRA authorized to access the file. This is only present if the value in the DRF offset field is non-zero.

The structure of a key list is shown below. This structure is used for both the DDF and DRF key lists in the EFSRPC Metadata.



Length of Key List: The number of entries in this key list. It is a 32-bit unsigned integer.

Key List Entry #1, #2, and so on: A number of entries equal to the value in the length of key list field. The format of the individual entries is specified below.

The format of an individual key list entry is as follows:

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
Length																															
Offset to Public Key Information																															
Encrypted FEK length																															
Offset to Encrypted FEK																															
Flags																															
Data Fields (variable)																															

Length: The length of this key list entry in bytes. It is a 32-bit unsigned integer.

Offset to Public Key Information: The offset to the Public Key Information in bytes from the start of this entry. It is a 32-bit unsigned integer. The Public Key Information is completely contained inside the Data Fields.

Encrypted FEK Length: The length of the data in the Encrypted FEK, in bytes. It is a 32-bit unsigned integer.

Offset to Encrypted FEK: The offset to the Encrypted FEK, in bytes from the start of this entry. It is a 32-bit unsigned integer. The Encrypted FEK is completely contained inside the Data Fields.

Flags: This field is used to indicate the algorithm used to encrypt the FEK in this key list entry. It is a 32-bit unsigned integer.

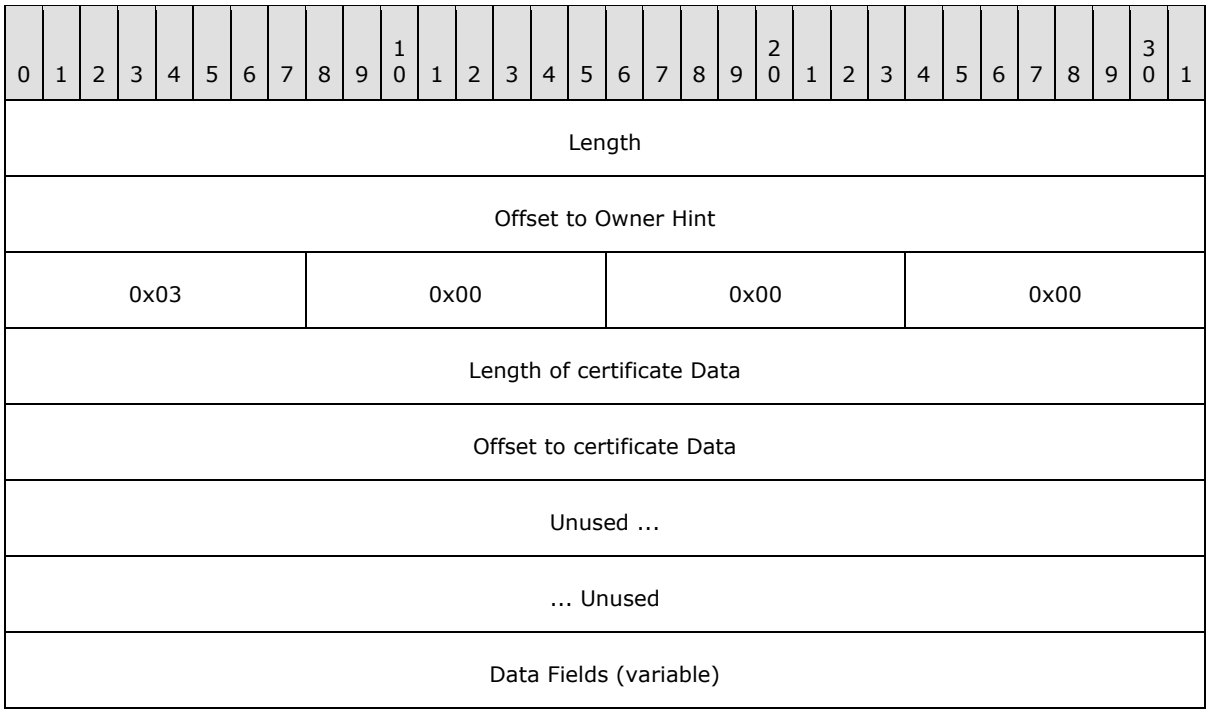
Value	Meaning
0x00000000	The Encrypted FEK field is encrypted using RSA, with a public key belonging to a user or DRA.
0x00000001	The Encrypted FEK field is encrypted using AES-256, with a key that is obtained by signing the non-terminated Unicode string "MICROSOFTE" (20 bytes long) with the user's RSA and computing the SHA-256 hash of the result. This value is used when a user's private key is stored on a smart card, to improve performance by minimizing the number of smart card accesses. This value is only used by Windows Vista and Windows Server 2008, and only in the DDF. All previous versions of Windows set this field to 0 when creating the EFSRPC metadata, ignore its value when processing the metadata, and expect the FEK to be encrypted using RSA.

Data Fields: This field contains the following items in any order at the locations indicated by the respective Offset fields above. These items are completely contained inside this field and do not overlap each other. There will be no unused areas within this field spanning more than eight contiguous bytes.

Public Key Information: The structure of this field is specified below. This field contains information about the X.509 certificate that contains the RSA public key which is used to encrypt the encrypted FEK field.

Encrypted FEK: The structure of this field is specified below. This field contains information about the FEK. It is encrypted as indicated by the contents of the Flags field.

The structure of the Public Key Information is as follows:



Length: The length, in bytes, of this structure. It is a 32-bit unsigned integer.

Offset to Owner Hint: If the Owner Hint is present, this field represents the offset of the Owner Hint from the beginning of this structure, measured in bytes. If this field is 0, then the Owner Hint is not present. This field contains a 32-bit unsigned integer.

Length of Certificate Data: The size, in bytes, of the Certificate Data. It is a 32-bit unsigned integer.

Offset to Certificate Data: The offset, in bytes, of the Certificate Data from the start of this structure. It is a 32-bit unsigned integer.

Unused: This field is set to 0 when the EFSRPC Metadata is created, and is ignored by the server.

Data Fields: This field contains the following items, in any order, and at the locations indicated by the respective Offset fields above. These items are completely contained inside this field and do not overlap each other. There will be no unused areas within this field that span more than eight contiguous bytes.

Owner Hint: A security identifier (SID) that is intended to be used as a hint regarding the identity of the key owner. This item is present only if the Offset to Owner Hint field is nonzero. The structure of a SID is specified in [MS-DTYP] section **2.5.2**.

Certificate Data: The structure of this item is specified below. Certificate Data provides information about the X.509 certificate associated with the public key that is used to encrypt the FEK data in this key list entry.

The structure of the Certificate Data is as follows:

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
Offset to Certificate Thumbprint																															
Length of Certificate Thumbprint																															
Offset of Container Name																															
Offset of Provider Name																															
Offset of Display Name																															
Data Fields (variable)																															

Offset to Certificate Thumbprint: Offset of the Certificate Thumbprint from the start of this structure. It is a 32-bit unsigned integer.

Length of Certificate Thumbprint: The length of the Certificate Thumbprint. It is a 32-bit unsigned integer.

Offset of Container Name: Offset of the Container Name, (in bytes) from the start of this structure. It is a 32-bit unsigned integer. If this field is set to 0, then the Container Name is absent.

Offset of Provider Name: Offset of the Provider Name, (in bytes) from the start of this structure. It is a 32-bit unsigned integer. If this field is set to 0, the Provider Name is absent. If a Provider Name is present, a Container Name will also be present.

Offset of Display Name: Offset of the Display Name, (in bytes) from the start of this structure. It is a 32-bit unsigned integer. If this field is set to 0, then the Display Name is absent.

Data Fields: This field contains the following items, in any order, and at the locations indicated by the respective Offset fields above. These items are completely contained inside this field and do not overlap each other. There will be no unused areas within this field that span more than eight contiguous bytes.

Certificate Thumbprint: The SHA-1 hash of the DER-encoded form of the certificate. For more information on SHA-1, see [FIPS180]. For more information on DER encoding, see [X690].

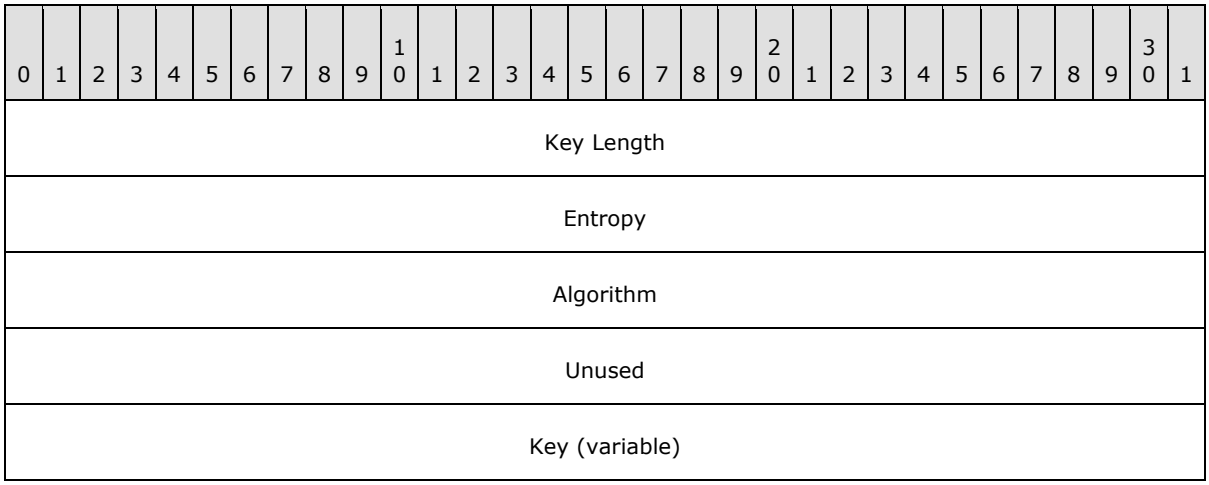
Container Name: A null-terminated Unicode string in UTF-16 encoding that provides a hint as to the public key container in which the key is stored. This item is always present if the Provider Name is present. It is only present if the Offset of Container Name field is nonzero. For more information about container names, see [MSDN-CSPCTX].

Provider Name: A null-terminated Unicode string in UTF-16 encoding that provides a hint as to the Cryptographic Services Provider in which the key is stored. This field is always present if the

Container Name is present. It is omitted if the Offset of Provider Name field is 0. For more information about Cryptographic Service Providers, see [MSDN-CSPR].

Display Name: A null-terminated Unicode string in UTF-16 encoding that provides a hint as to the friendly name that can be used to identify this certificate for display purposes. This field is omitted if the Offset of Display Name field is 0.

The Encrypted FEK field in the DDF and DRF key list entries consists of the following structure, encrypted as specified in the description of the Flags field for the key list entry. A Windows EFSRPC server will return an error if the total length of this structure exceeds 1,086 bytes.



Key Length: The length, in bytes, of the **Key** field. It is a 32-bit unsigned integer. Possible values depend on the Algorithm as specified in section 2.2.14.

Entropy: The number of bits of true randomness in the key contained in this structure. It is a 32-bit unsigned integer. Possible values depend on the Algorithm, as specified in section 2.2.14.

Algorithm: The symmetric cryptographic algorithm associated with this key. It is a 32-bit unsigned integer. Possible values are specified in section 2.2.14. The possible values for this field are constrained by the value of the EFS version field in the EFSRPC Metadata. The possible values for this field are constrained by the value of the EFS version field in the EFSRPC Metadata.

Unused: This field is set to 0 at creation time, and ignored when the EFSRPC Metadata is processed by the server.

Key: The FEK for the file.

<8> Section 2.2.3: The structure used by Windows for the EFSRPC Raw Data Format is shown below. A Windows EFSRPC server will return an error if the data passed to it by the **EfsRpcWriteFileRaw** method is not in this format.

The overall structure of the EFSRPC Raw Data Format is as follows:

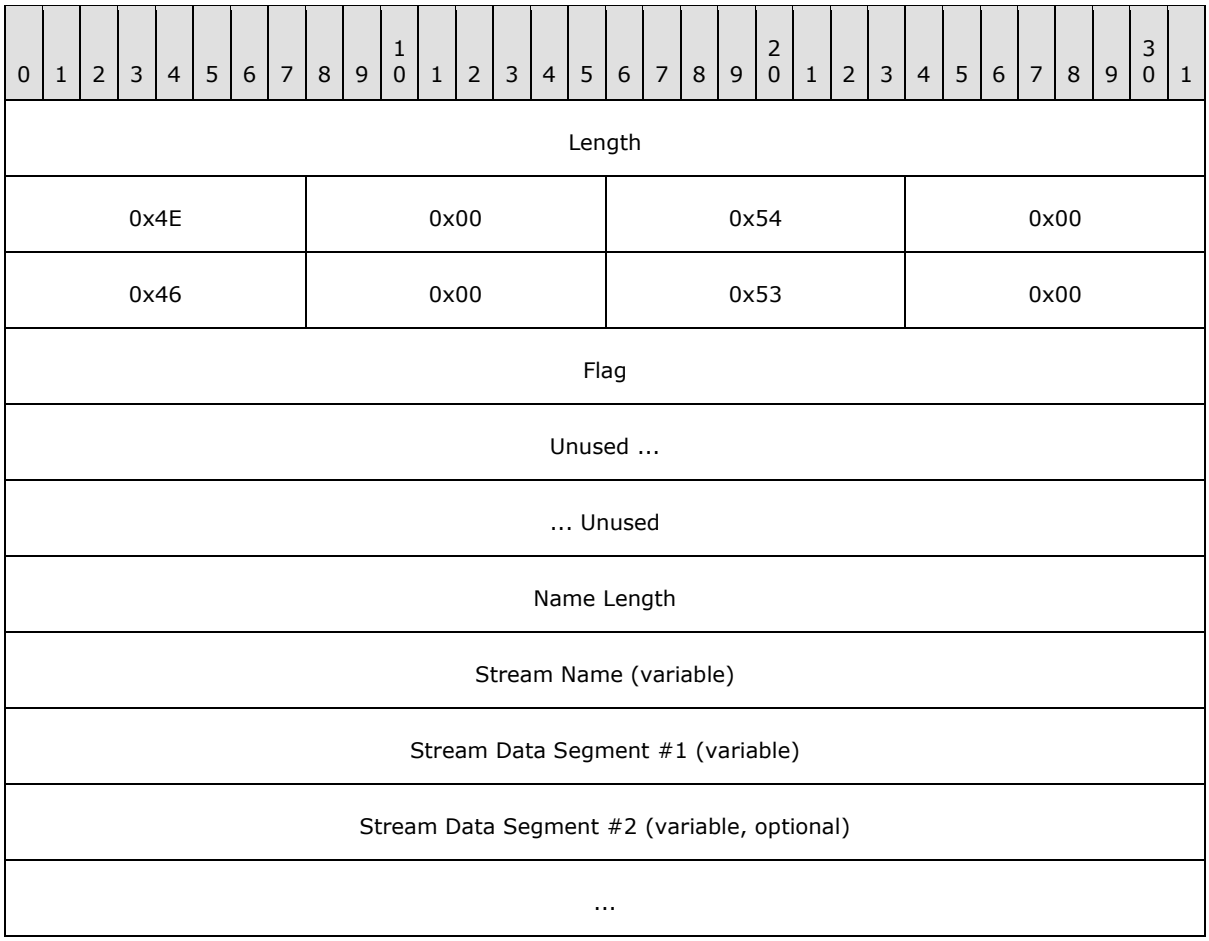
0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
0x00								0x01								0x00								0x00							
0x52								0x00								0x4F								0x00							
0x42								0x00								0x53								0x00							
Unused ...																															
... Unused																															
EFSRPC Metadata Stream (variable)																															
Additional Stream #1 (variable, optional)																															
Additional Stream #2 (variable, optional)																															
...																															

Unused: This field is always set to 0 and ignored.

EFSRPC Metadata Stream: The structure of this field is specified below. This field contains the EFSRPC Metadata for the file, along with a header. The structure of the EFSRPC Metadata is specified in section 2.2.2.

Additional Stream #1, #2, and so on: These correspond to marshaled versions of all the **streams** (except for EFSRPC Metadata) in the given file. They are optional and might not exist (for example, for folders with no alternate streams). For more information on NTFS file streams, see [MSFT-NTFS].

The structure of a marshaled stream is shown below, with notations for the special case of the EFSRPC Metadata stream.



Length: The length, in bytes, of this stream header from the start of this field to the end of the Stream Name field. It is a 32-bit unsigned integer.

Flag: This is a 32-bit unsigned integer. It is set to 0x00000000 if the stream data is encrypted by EFS. The **Flag** field is otherwise set to 0x00000001. The **Flag** field is always set to 0 in the case of the EFSRPC Metadata stream, and ignored by the server in that case.

Unused: This field is set to 0 and ignored.

Name Length: The length, in bytes, of the **Stream Name** field. It is a 32-bit unsigned integer. For the EFSRPC Metadata stream, this is always set to 0x00000002. In the case of the main (unnamed) data stream, this is equal to 0.

Stream Name: The name of the stream. This is set to either a null-terminated Unicode string in UTF-16 encoding, or an integer value stored in binary form. For the EFSRPC Metadata stream, this is always set to 0x1910. This field is absent for the main (unnamed) data stream of a file.

Stream Data Segment #1, #2, and so on: These segments contain the contents of the stream as well as some metadata for reassembling the segments. For encrypted streams, these segments also contain some metadata to aid in decryption.

The structure of the stream data segment is as follows:

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
Length																															
0x47								0x00								0x55								0x00							
0x52								0x00								0x45								0x00							
Unused																															
Data Segment Encryption Header (variable, optional)																															
Stream Data (variable)																															

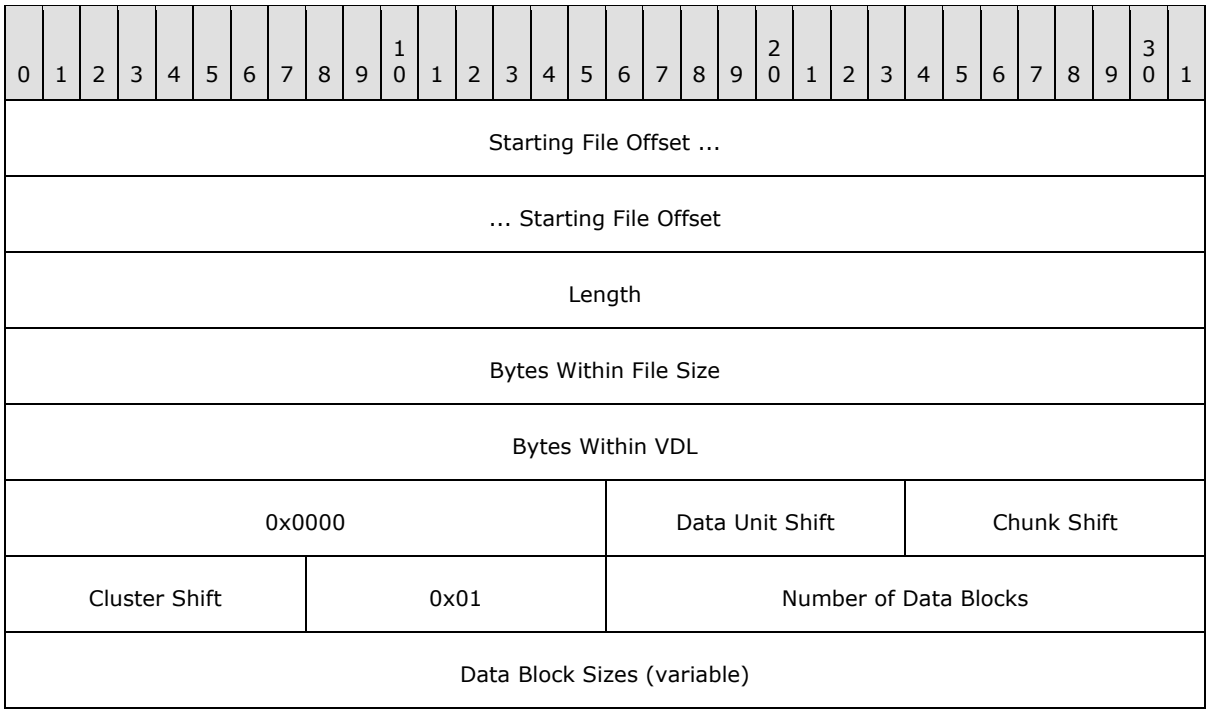
Length: The length, in bytes, of this segment. It is a 32-bit unsigned integer. The length is measured from the start of this field to the end of the **Stream Data** field.

Unused: This field is set to 0 and is ignored by the server.

Data Segment Encryption Header: This header is present only if the stream is encrypted (that is, if the **Flag** field in the stream header is set to 0 and this is not the EFSRPC Metadata stream). The details of the Data Segment Encryption Header structure are specified below.

Stream Data: This field contains part or all of the stream data. If the Data Segment Encryption Header is present, Stream Data will be consistent with it. Stream Data consists of contiguous bytes taken from the stream except for zero bytes that are omitted in accordance with the Data Segment Encryption Header.

The structure of the Data Segment Encryption Header is as follows:



Starting File Offset: This field contains an unsigned 64-bit integer denoting the offset, in bytes, into the stream being serialized of the first data byte contained in this data segment.

Length: The length of this header, in bytes, measured from the beginning of the **Starting File Offset** field to the end of the Data Block Sizes field. It is a 32-bit unsigned integer.

Bytes Within Stream Size: The number of bytes contained within this stream data segment that fall within the stream size. It is a 32-bit unsigned integer. This may be less than the number of bytes actually present due to padding required by the encryption algorithm.

Bytes Within VDL: The number of bytes contained within this stream data segment that fall within the **valid data length (VDL)**. It is a 32-bit unsigned integer. This may be less than the number of bytes actually present due to padding required by the encryption algorithm. Bytes beyond the VDL MUST be set to 0 after decryption.

Data Unit Shift: The base-2 logarithm of the data unit size. It is an 8-bit unsigned integer. For files that are not **sparse files**, the data unit size is simply the size of the data in this segment. For sparse files, it is the size of a compression unit, which is the smallest unit that all holes MUST be a multiple of.

Chunk Shift: The base-2 logarithm of the chunk size. It is an 8-bit unsigned integer. The chunk size is equal to the data unit size.

Cluster Shift: The base-2 logarithm of the cluster size in bytes. It is an 8-bit unsigned integer. The cluster is the smallest unit of allocation in the NTFS.

Number of Data Blocks: This field contains the number of data blocks specified in this segment. It is a 16-bit unsigned integer. It is equal to the number of entries in the Data Block Sizes field specified next.

Data Block Sizes: This field consists of a sequence of unsigned 32-bit values denoting the sizes of the successive data blocks in the Stream Data field that follows this header. Each value in the sequence is less than or equal to the data unit size, unless it spans the VDL or a hole in the case of a sparse file.

<9> Section 2.2.3: Windows 2000, Windows XP, Windows Server 2003 and the RTM version of Windows Vista throw an RPC_PIPE_DISCIPLINE_ERROR (0x077D) exception in this case. Windows Vista SP1 and Windows Server 2008 OS versions return a nonzero error.

<10> Section 2.2.7: The Windows implementation of the EFSRPC server returns an error if the size of this encoded certificate exceeds 32 kilobytes. This restriction is represented in the range attribute of cbData.

<11> Section 2.2.8: As a defensive measure against overflow attacks, the Windows implementation of the EFSRPC server restricts the size of the **bData** field to 100 bytes, and returns an error if this size is exceeded. This restriction is represented by the range attribute of cbData.

<12> Section 2.2.10: As a defensive measure against overflow attacks, the Windows implementation of the EFSRPC server restricts the number of entries in this array to 500, and returns an error if this size is exceeded. This restriction is represented in the range attribute of nUsers.

<13> Section 2.2.12: As a defensive measure against overflow attacks, the Windows implementation of the EFSRPC server restricts the number of entries in this array to 500, and returns an error if this size is exceeded. This restriction is represented in the range attribute of nCert_Hash.

<14> Section 2.2.13: As a defensive measure against overflow attacks, the Windows implementation of the EFSRPC server restricts the size of this object to 260 kilobytes, and returns an error if this size is exceeded. This restriction is represented in the range attribute of cbData.

<15> Section 2.2.14: Windows 2000 supports both versions of CALG_DESX, and Windows XP supports both versions of CALG_DESX and CALG_3DES. Windows XP SP1 adds support for CALG_AES_256; Windows Server 2003, Windows Vista, and Windows Server 2008 support all the algorithms shown.

<16> Section 2.2.15: The combinations of values for the **Entropy**, **Algorithm**, and **KeyLength** fields used by Windows are summarized in the following table. A Windows server will return an error if these fields are not set to one of the following combinations.

Algorithm used	Algorithm	Entropy	Keylength
CALG_AES_256	0x6610	256	32
CALG_3DES	0x6603	168	24
CALG_DESX (domestic)	0x6604	128	16
CALG_DESX (export)	0x6604	56	16

The value of the **Entropy** field is less than eight times the value of the **KeyLength** field in the following cases.

- For CALG_3DES, the difference between entropy and key length is due to the parity bits included in the key. For more information, see [TDEA].

- For CALG_DESX (export), the difference is because only the first 56 bits are random and the rest are set to 0.

<17> Section 3: The Windows Vista and Windows Server 2008EFSRPC client implementations attempt to negotiate the RPC_C_AUTHN_LEVEL_PKT_PRIVACY option with RPC_C_AUTHN_GSS_NEGOTIATE. If an error is encountered and the client is not configured to require security, the client falls back to connecting over an insecure RPC connection. Versions of Windows up to and including Windows XP SP2 do not support this option; they neither register any SSPI providers on the server side, nor do they request any on the client.

<18> Section 3.1.3: Windows Vista and Windows Server 2008 both register an SSPI provider to support RPC_C_AUTHN_LEVEL_PKT_PRIVACY.

<19> Section 3.1.4: Windows implementation uses the RPC protocol to retrieve the identity of the caller, as described in [MS-RPCE] section 3.3.3.4.3. The server uses the underlying Windows security subsystem to determine the permissions for the caller. If the caller does not have the required permissions to execute a specific method, then the method call fails with ERROR_ACCESS_DENIED.

<20> Section 3.1.4: Windows 2000 supports methods with opnums 0 through 11. For opnum 11, it behaves as the **EfsRpcDuplicateEncryptionInfoFile** method . Windows XP and Windows Server 2003 support opnum 0 through opnum 13, with opnum 11 being implemented as the **EfsRpcNotSupported** method. Windows Vista and Windows Server 2008 support all the methods specified in this document. They implement opnum 11 as the **EfsRpcNotSupported** method.

<21> Section 3.1.4: Gaps in the opnum numbering sequence apply to Windows as follows:

Opnum	Description
10	Only used locally by Windows, never remotely.
14	Only used locally by Windows, never remotely.
17	Only used locally by Windows, never remotely.

<22> Section 3.1.4: The error codes returned by the Windows implementation of EFSRPC are specified in [MS-RPCE] section 3.

<23> Section 3.1.4: All Windows implementations check that any file name arguments passed refer to local files, and that the calling user has the appropriate NTFS access permissions on the local file system.

<24> Section 3.1.4.1: If the CREATE_FOR_DIR flag is set, and the CREATE_FOR_IMPORT flag is also set, Windows will attempt to create a folder with the specified name instead of a file. Note that the CREATE_FOR_DIR flag is not used with the CREATE_FOR_IMPORT flag when restoring a file because errors will result when trying to write file data to the resulting folder.

<25> Section 3.1.4.1: Windows 2000, Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008 support all the flags specified here.

<26> Section 3.1.4.1: Windows EFSRPC servers will return an error unless at least one of the following is true:

- The calling user has a private key that grants him authorized access to the file.
- The CREATE_FOR_IMPORT flag is set, and the user has restore rights on the server.

- The CREATE_FOR_IMPORT flag is not set, and the user has backup rights on the server.

<27> Section 3.1.4.1: When this flag is set, a Windows server will overwrite an existing hidden file with the same name. If this flag is not set, it will return an error if a hidden file of the same name already exists.

<28> Section 3.1.4.1: If the CREATE_FOR_IMPORT flag is not set, Windows implementations ignore the value of the CREATE_FOR_DIR flag except if the FileName argument refers to a file, in which case an error (ERROR_DIRECTORY, 0x10B) is returned.

<29> Section 3.1.4.1: In Windows, when the CREATE_FOR_IMPORT flag is set, the server overwrites all files, except for hidden files, by default. If the Flags parameter has the OVERWRITE_HIDDEN bit set, the server also overwrites hidden files.

<30> Section 3.1.4.2: Windows 2000, Windows XP, Windows Server 2003, and the RTM version of Windows Vista throw an RPC_PIPE_DISCIPLINE_ERROR (0x077D) exception in this case. Windows Vista and Windows Server 2008 return a nonzero error.

<31> Section 3.1.4.2: Windows 2000, Windows XP, Windows Server 2003, and the RTM version of Windows Vista throw an RPC_PIPE_DISCIPLINE_ERROR (0x077D) exception in this case. Windows Vista and Windows Server 2008 return a nonzero error.

<32> Section 3.1.4.3: Windows 2000, Windows XP, Windows Server 2003, and the RTM version of Windows Vista throw an RPC_PIPE_DISCIPLINE_ERROR (0x077D) exception in this case. Windows Vista and Windows Server 2008 return a nonzero error.

<33> Section 3.1.4.3: Windows 2000, Windows XP, Windows Server 2003, and the RTM version of Windows Vista throw an RPC_PIPE_DISCIPLINE_ERROR (0x077D) exception in this case.Windows Vista and Windows Server 2008 return a nonzero error.

<34> Section 3.1.4.5: Windows servers always return success for folders. For files, they return ERROR_ACCESS_DENIED if the file is already encrypted and the calling user does not have access to a private key that can decrypt the file.

<35> Section 3.1.4.5: When the FileName parameter refers a file, Windows implementations will encrypt the file, create EFSRPC Metadata for it, and store the metadata with the file. When it refers to a folder, Windows implementations will create EFSRPC Metadata for the folder, and set an attribute on the folder that instructs NTFS to encrypt any new files created in that folder. Existing files within the folder will not be affected in any way.

<36> Section 3.1.4.6: When the *FileName* parameter refers a file, Windows implementations will decrypt the file and discard its EFSRPC Metadata.This action will only succeed if the calling user has access to a private key that will decrypt the file.

When *FileName* refers to a folder, Windows implementations will discard EFSRPC Metadata for the folder, and unset the attribute on the folder that instructs NTFS to encrypt any new files created in that folder. Existing files within the folder will not be affected in any way. This action will always succeed.

<37> Section 3.1.4.7: Windows implementations behave identically with files and folders. In both cases, they return the list of users contained in the EFSRPC Metadata.

<38> Section 3.1.4.8: Windows implementations behave identically with files and folders. In both cases, they return the list of DRAs contained in the EFSRPC Metadata.

<39> Section 3.1.4.9: Windows servers will return an error if the calling user does not possess a private key corresponding to a user certificate that is authorized to access the file.

<40> Section 3.1.4.9: Windows implementations behave identically with files and folders. In both cases, they modify the EFSRPC Metadata on the file or folder.

<41> Section 3.1.4.10: Windows implementations behave identically with files and folders. In both cases, they modify the EFSRPC Metadata on the file or folder.

<42> Section 3.1.4.11: Windows 2000 responds to this opnum, as described in section 3.1.4.13. Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008 return an error.

<43> Section 3.1.4.11: Windows 2000 performs the processing specified in section 3.1.4.13. All other versions of Windows return ERROR_NOT_SUPPORTED.

<44> Section 3.1.4.12: UPDATE_KEY_USED is supported on Windows Server 2003. BASIC_KEY_INFO is supported on Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. The other flags are only supported on Windows Vista and Windows Server 2008.

<45> Section 3.1.4.12: Windows implementations behave similar to files and folders. In both cases, they return information about the FEK contained in the EFSRPC Metadata on the file or folder.

<46> Section 3.1.4.12: Windows Server 2003 requires that UPDATE_KEY_USED only be used with a folder name. If not, an error is returned.

<47> Section 3.1.4.12: Windows Server 2003 updates the EFSRPC Metadata of that all the encrypted files and folders accessible by the calling user in the *FileName* parameter or one of its subfolders to use a single user certificate for that user.

<48> Section 3.1.4.12: Windows implementations behave identically in both cases with files and folders. No additional restrictions are placed on CHECK_ENCRYPTION_STATUS. In the case of CHECK_DECRYPTION_STATUS, Windows implementations return ERROR_REQUIRES_INTERACTIVE_WINDOWSTATION (0x05b3) if the caller is not logged on locally.

<49> Section 3.1.4.13: On Windows, this value is a bitwise OR of zero or more of the following values. Alternatively, it can be set to FILE_ATTRIBUTE_NORMAL (0x00000080), signifying that no other attributes are to be set.

Value	Meaning
FILE_ATTRIBUTE_HIDDEN 0x00000002	The file is hidden (not displayed in normal folder listings).
FILE_ATTRIBUTE_ARCHIVE 0x00000020	This attribute is used by applications to mark files for backup or removal.
FILE_ATTRIBUTE_TEMPORARY 0x00000100	The file is being used for temporary storage.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED 0x00002000	The file's contents should not be indexed by the content indexing service.
FILE_ATTRIBUTE_NORMAL 0x00000080	No other attributes are to be set.

<50> Section 3.1.4.13: The data portion of the EFS_RPC_BLOB structure is expected to contain a security descriptor. This is an opaque data type in Windows that is best manipulated only indirectly through the APIs provided for this purpose, as specified in [MS-DTYP] sections 2.4.6 and 2.5.

<51> Section 3.1.4.13: Windows clients always set this parameter to FALSE. If the *bInheritHandle* parameter is set to TRUE, a Windows server will open the file referred to by the *DestFileName* parameter such that the resulting file handle can be inherited by a child process of the server. However, this behavior does not affect the protocol or the client's behavior.

<52> Section 3.1.4.13: The **EfsRpcDuplicateEncryptionInfoFile** method was associated with opnum 11 in Windows 2000; therefore, it cannot be used between a Windows 2000 client and a server running a different version of Windows, or vice versa.

<53> Section 3.1.4.13: Windows implementations behave identically with files and folders. In both cases, they modify the EFSRPC Metadata on the destination file or folder. If the *SrcFileName* parameter refers to a file, the *DestFileName* parameter refers to a file as well, and similarly for folder; otherwise, an error is returned. If a new object needs to be created, then it is created as a file or folder depending on the type of *SrcFileName*.

<54> Section 3.1.4.13: Windows Server 2008 servers will generate a new FEK for the *DestFileName* parameter if only one user has access to the *SrcFileName* parameter. This is to ensure that users who get access to one of the two files at a later date do not automatically get the ability to decrypt the other file.

<55> Section 3.1.4.14: For smart card certificates, if the private key is available and if permitted by the prevailing policy, the added entry will use **AES**-256 encryption with a derived key instead of RSA encryption with the public key, as described in the Windows Behavior note for section 2.2.2 in Appendix B.

<56> Section 3.1.4.14: Windows implementations behave similar to files and folders. In both cases, they manipulate the EFSRPC Metadata on the file or folder.

<57> Section 3.1.4.15: UPDATE_KEY_USED is supported on Windows Server 2003. BASIC_KEY_INFO is supported on Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. The other flags are only supported on Windows Vista and Windows Server 2008.

<58> Section 3.1.4.16: Windows returns the metadata in the format described in the Windows Behavior note for section 2.2.2 in Appendix B.

<59> Section 3.1.4.16: Windows implementations behave similar to files and folders. In both cases, they return the EFSRPC Metadata on the file or folder.

<60> Section 3.1.4.17: Windows servers expect the metadata to conform to the format described in the Windows Behavior note for section 2.2.2 in Appendix B, and will return an error if this is not so.

<61> Section 3.1.4.17: Windows servers expect the metadata to conform to the format specified in Windows Behavior note 6 in section 7, and will return an error if this is not so.

<62> Section 3.1.4.17: Windows implementations behave similar to files and folders. In both cases, they modify the EFSRPC Metadata on the file or folder.

<63> Section 3.1.6: Windows servers de-allocate connection resources upon receiving an indication of a broken connection to the client. If an error is encountered while processing the **EfsRpcReadFileRaw** (3.1.4.2) method, the server will return an error and flush the pipe by performing a zero-byte write operation to it.

8 Index

T

V

W