OSR File Encryption Solution Framework

# Solution Developer's Guide

# V1.5

# 1 Introduction

This guide provides the necessary architectural and reference information to allow Client developers to design and interface a Solution with the OSR File Encryption Solution Framework (FESF). A "Client" in this context means a licensed user of the FESF product. A "Solution" is a group of one or more Client-developed programs that, combined with FESF, perform or utilize on-access per-file encryption services. Solutions might range in scope from a basic file encryption product to a document management system that includes per-file encryption as a small part of a much more comprehensive product suite.

This guide seeks to provide the conceptual background and terminology necessary to allow you to successfully design and build your Solution using FESF. The guide also contains reference material for the callbacks from FESF to your Solution and the support routines provided by FESF to making writing your Solution easier.

The reader is assumed to be a C/C++ system programmer who is familiar with general Windows architectural concepts such as security and common Windows programming concepts such as COM.

# 2 FESF Overview and Basic Concepts

The OSR File Encryption Solution Framework (FESF) allows Clients to incorporate transparent, on-access, per-file encryption into their products. While adding on-access encryption sounds like something that should be pretty simple to accomplish, it turns out to be something that's exceptionally complicated. Creating a Solution that performs well is even more difficult.

FESF handles most of the necessary complexity, including the actual encryption operations, in kernel mode. This allow Clients to build customized file encryption products with no kernel-mode programming.

To understand what needs to be created to transform FESF into a complete product, it's important to understand a few concepts that are central to FESF. We discuss those concepts in this section of this document.

## 2.1 Policy

When we talk about "Policy" in FESF, we mean exactly two things:

- Whether a newly created file should contain Client defined control information and FESF metadata information and whether data written to that newly created file should be transparently encrypted before being stored on disk.

- Whether the data read from an existing FESF encrypted file should be decrypted before being returned to the application that's reading it and whether data written by that application should be encrypted before it's stored in that existing FESF encrypted file on disk.

Policy decisions are made in user-mode by the Client Solution. The Client's Solution code is called to make a Policy decision whenever (a) a new file is created (and before any data is written to it), or (b) an existing FESF encrypted file is opened (and before any data has been read from or written to it). Although there are a few additional events that will result in FESF calling the Client Solution, these are the only times when FESF calls the Client Solution to ask for a Policy decision.

The Client Solution will use data provided by FESF and optionally other data it collects or maintains independent of FESF as the basis for its policy decision. When a Policy decision is required, FESF provides the following information to the Solution:

- The fully-qualified path of the file being created or accessed. Specifically:
    - For files on local volumes, this includes the Volume GUID identifying the volume on which the file resides, plus the directory path and file name. The Volume GUID unambiguously identifies the volume, and can be converted to a drive letter by an FESF-supplied utility function.
    - For files on shadow volumes, the Volume GUID is predefined as being FE_SHADOW_VOLUME_GUID. The device name of the shadow volume is supplied as well as the volume relative path to the file.
    - For files on the network, the Volume GUID is predefined as being FE_NETWORK_GUID. The server and share are supplied along with the share relative path to the file.

- The Thread ID (TID) of the thread that's creating or accessing the file. Given this TID, the Solution can determine the fully-qualified path of the executing image and the Security ID (SID) under which the application is executing. The SID identifies a user (including username and domain) or other Windows security principal (such as a security group). FESF provides utility functions to allow the Solutions to easily determine these values from the provided TID.

- The file access (read data, write data, and others) that was granted to the accessing application.

- The disposition of the file being accessed. This indicates whether the file is being created, overwritten, appended, or just opened.

Thus, when a new file is created or an existing encrypted file is accessed, the Client Solution determines policy for that file based on some combination of:

- Drive (or server and share), directory path, and name of the file being accessed
- Path and name of the accessing application
- Security context (that is, nominally the user) under which the application is running.
- The action being performed on the file.
- The access granted to the application for this specific open instance of the file.

Which of these variables are taken into account, and how they may be used to define Policy, is entirely up to the Client Solution. In addition, variables other than those provided directly by FESF – such as the system on which the application is running, or the day of the week – could also be used.

By way of example, a very simple policy implemented by a Client's Policy DLL might be:

> "We want any files that are created in the directory \MySecretStuff\ on the volume that is the C drive on this workstation to be encrypted."

That's pretty straight forward. Or, a slightly more involved example:

> "Decrypt all encrypted files on the network with the path \\SpecialForces\missions\ImpossibleMission\ only when they are accessed by a user that's in the Active Directory security group *SecretAgents* and from a system that is actively joined to a domain named *MI5* or *MI6.*"

That's also pretty simple, but requires the Solution to get the name of the domain to which the machine is joined outside the mechanisms provided by FESF (it's easy: You just call the Windows function *LsaQueryInformationPolicy*). A slightly more complex policy that a Solution could implement would be:

> "We want all existing encrypted files accessed by Microsoft Outlook, regardless of the directory that the file may be in, to not be decrypted when Outlook accesses it, unless the file has the file suffix OST or PST."

This third example policy would ensure that if a user attached an encrypted document to an Outlook email, the encrypted version of the file would be sent, while still allowing locally stored Outlook data files (the OST and PST files) to be encrypted.

## 2.2 Policy Definition and Storage

So where and how is Policy defined? Once it's been defined where and how is it stored? These are both entirely under the control of the Client Solution. In terms of definition and storage, the only thing that is important to FESF is that the Client Solution promptly responds to callbacks from FESF when FESF asks it for Policy decisions.

A Solution's Policy could be defined by a GUI program or even an MMC snap-in developed as part of the Solution. Because the factors that are used to define Policy are determined by the Client Solution, FESF does not provide any standard mechanism for defining Policy.

Some Solutions may store the Policy information in a proprietary Policy server. Others might encode the information and store it in the Active Directory, using custom extensions of the AD schema. Because the format and content of Policy is entirely defined by the Client Solution, FESF does not require (or provide) any standard location for Policy storage.

## 2.3 Communicating Policy from Client Solution to FESF

When FESF wants to know the Policy for a given access operation on a particular file, it calls a callback in the Client Solution. The entity within FESF that performs this callback is the FESF Policy Service (FesfPolicy). FesfPolicy is a standard Windows user-mode service. The callback function that FesfPolicy calls is provided by the Solution in a DLL known as the Client Policy DLL. FesfPolicy loads this DLL dynamically when it starts based on a Registry parameter.

We'll describe a great deal more about the Client Policy DLL later in this document. However, what's important to understand at this point is that the Policy DLL is the (one and only) way that FESF asks the Client Solution for Policy decisions. Thus, the Client Policy DLL is **the** interface between FESF and the Client Solution when it comes to determining Policy for a file.

For example, each time a new file is created on a system with FESF running, FesfPolicy will call the Policy DLL's *PolGetPolicyNewFile* callback function. As the return value from this function, the Client Policy DLL indicates whether data should be encrypted when written to the file that's being created or whether data should be written to the file being created as clear text. Similarly, each time an existing encrypted file is opened, the Policy DLL's *PolGetPolicyExistingFile* callback function is invoked. And, similarly, the return value from this function indicates whether FESF should transparently encrypt/decrypt data when this application instance writes/reads the file, or whether FESF should provide "raw" access (that is, access without transparent encryption or decryption).

## 2.4 Key Material and Encryption Identification

As previously described, each time a new file is created, the Client Solution Policy DLL is called by FESF. If the Policy DLL indicates that data written to the newly created file should be encrypted, the Solution returns three things to FESF:

1. Header Data: This data – which is entirely defined by the Client Solution -- will be stored by FESF exactly as provided in the newly created file. This Header Data will be provided by FESF to the Client Solution whenever the file is subsequently opened and the key is required. The Header Data may contain any information useful to the Solution, with the restriction that having determined decrypted access is desired, the Solution must be able to derive the key data for the file given this Header Data.
2. Algorithm ID: This indicates which encryption algorithm (and associated properties) FESF will use to encrypt/decrypt the file's data.

3. Key: The key data to be used to encrypt and/or decrypt the file's data.

When an existing encrypted file is opened, the Client Solution Policy DLL is called with the path of the file being opened and the Header Data that was previously stored in the file (along with other data). This Header Data was supplied by the Solution when the file was created. Using this Header Data, the Client Solution is responsible for returning an Algorithm ID and Key Data for FESF to use to decrypt the file's data and encrypt any data that may be subsequently written to the file.

## 2.5 Where is the Encryption Actually Done and What Algorithms Are Supported?

In the course of normal operations, encryption and decryption are performed in kernel-mode under FESF's control. However, FESF itself does not include any encryption components or algorithms. Rather, FESF calls Microsoft's Cryptography API: Next Generation (CNG) package to accomplish the actual encryption and decryption operations. CNG includes support for several standard algorithms (including DES, DESX, 3DES, RC2, RC4, and AES) and multiple modes for each algorithm. In addition, custom CNG Cryptographic Algorithm Providers can be written by Clients to support any desired algorithm.

FESF is careful to handle key material securely in kernel mode. For example, kernel components never store key material in pageable memory and scrub the contents of memory used for key material storage prior to deallocation.

When FESF is not installed some encryption and decryption may be performed in user-mode with the assistance of FESF supplied Stand Alone library functions.

### 2.5.1 A Word About Encryption Block Size and Initialization Vectors

For algorithms requiring a fixed block size, we use a value of 256 bytes. This choice is arbitrary. Normally, algorithms that provide a CBC mode also include a non-secret value known as the *initialization vector*. This prevents identical blocks from appearing to be identical in the encrypted file content.

When calling CNG encryption methods that require an initialization vector (IV), FESF generates this from the key material using a technique adapted from the disk drive field and known as the Encrypted Salt-Sector Initialization Vector (ESSIV). More details about how FESF generates the IV can be found in the **FESFSa Function Reference** section, elsewhere in this document.

## 2.6 Existing Files Are Not Automatically Encrypted

A careful reader might notice that we have so far only described how *newly created files* are transparently encrypted by FESF and how *existing files* that are already encrypted are handled by FESF. We have not, however, discussed how existing files that are not encrypted become encrypted. In other words, continuing one of our previous example where we had the Policy:

> "We want any files that are created in the directory \MySecretStuff\ on the volume that's the C drive on this workstation to be encrypted."

Any files that are newly created in the directory \MySecretStuff\ would be automatically encrypted by FESF after this policy was established (based on the response received when the Solution's Policy DLL is called). But suppose some files already existed in the \MySecretStuff\ directory when this Policy was established. How would these files become encrypted?

The answer is, it is up to the Client Solution to request that those files be encrypted, if and when desired. This is because only the Client Solution understands when Policy can be defined or changed, what security risk is associated with having existing unencrypted files in various locations, how many files might need to be encrypted as a result of a new Policy being created, and when an appropriate time to encrypt affected files might be. Some Client Solutions

might require Policy to be defined network-wide, and then perform encryption of existing files on individual workstations at "pre-boot" startup time (before users are allowed to login to the system). Others might choose to never encrypt existing files. FESF provides complete flexibility in this regard.

## 2.7 The Basics of Policy Operation

With the background provided so far, we can now discuss more details about the flow of control for accessing files when FESF is installed.

### 2.7.1 Raw vs Encrypted/Decrypted Access to Newly Created Files

For each new file that's created, FESF calls the Client Solution's Policy DLL at its *PolGetPolicyNewFile* callback function to determine the Policy for that file. In other words, FESF calls the Policy DLL to determine whether the data written to the file should be encrypted. If the Solution indicates that the data should be encrypted, FESF next calls the Policy DLL's *PolGetKeyNewFile* to get the Header Data, Algorithm ID, and encryption Key data for the newly created file.

Using the provided Algorithm ID and Key, FESF transparently encrypts data written to the file and decrypts data read from the file. In addition, FESF adds control and consistency metadata information to the file, including the Client-defined Header Data, to enable later validation and decryption.

If the Policy DLL indicates the data should not be encrypted, FESF performs no additional processing on the file's data. The file's data is written without modification. The Policy DLL's *PolGetKeyNewFile* is not called, and FESF adds no additional information to this file.

### 2.7.2 Raw vs Encrypted/Decrypted Access to Existing Encrypted Files

For each existing encrypted file that's accessed, the Client Solution's Policy DLL is called at its *PolGetPolicyExistingFile* callback function to determine whether that particular open instance should be granted raw or encrypted/decrypted access

Open instances that receive encrypted/decrypted access result in file data being transparently decrypted by FESF when read, and transparently encrypted by FESF when written. This is the typical mode for "permitted" applications. Data is encrypted while stored (at rest) on disk, but applications transparently see ordinary (plaintext) data. To enable these transparent encryption/decryption operations, FESF calls the Solution's Policy DLL at its *PolGetKeyFromHeader* callback function. FESF passes the Policy DLL's Header Data that was previously returned by the Policy DLL's *PolGetKeyNewFile* callback when the file was created. Given this Header Data and the path of the file, the Policy DLL returns the Algorithm ID and Key.

Open instances that receive raw access see data without any additional processing by FESF. Raw access is typically given to programs such as backup utilities. This results in the backed-up data being restored in encrypted form.

## 2.8 FESF Policy Caching

A powerful feature of FESF is the FESF Policy Caching. A Solution's Policy DLL may enable FESF Policy Caching by setting the **AccessCache.Enable** field of the **FE_POLICY_CONFIG** structure to **true**. To ensure good system performance, we strongly recommend all Solutions enable Policy Caching. Windows applications, including system components such as the Windows shell (Explorer.exe), have a strong propensity to open and close files repeatedly. This is why Policy Caching is so critical.

When FESF Policy Caching has been enabled by a Policy DLL, FESF makes an entry in its kernel-mode Policy Cache for the file after it returns from each call to *PolGetPolicyNewFile* or *PolGetPolicyExistingFile*. The data stored in the FESF Policy Cache is based on the values passed into and returned by those functions, and includes:

- Accessing process. This is the process that owns the thread indicated in the *ThreadId* argument.

- Access. This is the value supplied in the *Granted Access* argument.
- FE_POLICY_RESULT. This is the return value from the Policy DLL.

Note that these cache entries are associated exclusively with a particular file that is being opened. Each subsequent time that same file is opened, FESF consults the FESF Policy Cache for the file. If an entry exists in the cache for the same process and the same type of access, FESF uses the cached FE_POLICY_RESULT instead of calling the Policy DLL. This eliminates the overhead of calling the Policy DLL to determine policy for a file, process, and access type when the Policy DLL has already returned the desired policy (for that file, policy and access type) to FESF. An exception to this behavior is when a thread is "impersonating" (that is using different security information than the process that owns the thread). The FESF Policy Cache is never consulted for files accessed by impersonating threads.

The duration of this caching behavior lasts as long as the file remains open or Windows retains file (data) caching information for the file, whichever is longer. On systems with lots of free memory, caching can persist for a very long time (many hours) after a file has been closed. On systems with significant memory pressure, caching might persist only as long as a thread actively has an open handle to a file.

While the life of the FESF Policy Cache cannot be extended arbitrarily, entries can be selectively removed from the FESF Policy Cache at any time by the Policy DLL. The Policy DLL can remove entries in the FESF Policy Cache for a chosen file by calling the FesfUtility function **PurgePolicyCacheFile**. This removes all FESF Policy Cache entries for a given file. The Policy DLL can also remove all entries from the FESF Policy Cache related to a given process. This is done by calling the FesfUtility function **PurgePolicyCacheThread**, and providing the Thread ID of any thread in the process. This call removes all FESF Policy Cache entries for all files for the process associated with the thread (or for all processes, if the provided TID is zero).

Finally, it should be noted that if the FESF Policy Service terminates or becomes unresponsive, the FESF Policy Cache is completely purged.

## 2.9 Online, Offline, and Not Installed

FESF is capable of operating in three states:

- **Online State** – In this state, the FESF Kernel Mode Components and FESF User Mode Components (including the FESF Policy Service) are operating. In addition, enough of the Client Solution is operating to return prompt Policy decisions. This is the ordinary state of FESF on a running system.

- **Offline State** – In Offline State, FESF Kernel Mode Components are installed and running, but the FESF User Mode Components and/or Client Solution are not operating. This state can occur when the user mode components terminate unexpectedly and have yet to restart fully. This state also applies during the earliest part of system startup. FESF Kernel Mode Components are loaded and started at Boot time, but (due to the way Windows works) the FESF User Mode Components start slightly later. The time between when the FESF Kernel Mode Components are active and the time that the FESF Policy Service can promptly return Policy decisions, FESF runs in Offline State. This state also occurs during the system shutdown process.

- **Not Installed State** – In this state, the FESF Kernel Mode Components are either not installed or are absolutely and definitively known to be not running and unavailable on the system. Therefore, these components cannot be used to provide support for any operations, including file encryption

In Online State, Policy decisions are made based on information provided by the Client Solution.

In Offline State, the FESF Policy Cache is purged and any new open requests are granted raw access. Rename and hard link creation requests are treated according to the defaults established by the Policy DLL during its initialization.

Read and write operations to encrypted files that are currently open with encrypt/decrypt will continue to receive this access (with data being transparently encrypted and decrypted) until the file is closed.

In Not Installed State, ordinary FESF operations are not available (obviously… because FESF is not installed). On systems in this state, FESF supports the use of Stand Alone Utilities that can be used to encrypt, decrypt, or change the Header Data of files. FESF currently supports developing Stand Alone Utilities on both Windows and Linux, through use of an FESF-provided library. This library (FESFsa.lib) interprets FESF On Disk Structure (ODS) and provides a highly flexible framework for performing encryption and decryption operations using Client-provided code. This library is described later in this document.

## 2.10 Files or Streams?

One final detail remains to be discussed. The FESF documentation, and even the names of interfaces, uniformly refers to files as the unit of access. For example, we might describe the *PolGetPolicyNewFile* function as follows:

> "A Policy DLL's PolGetPolicyNewFile callback function determines whether a new file should be created in encrypted or non-encrypted format."

While this is correct, it doesn't say anything about how FESF deals with alternate data streams ("streams") on file systems that support them.

FESF is fully stream aware. This means that FESF supports accessing, and optionally transparently encrypting and decrypting, data on a per-stream basis on file systems that support alternate data streams. Therefore, for file systems that support streams, the FESF documentation should be read as including "stream" whenever the term "file" is encountered.

On file systems that support alternate data streams, the file name information passed into the Client Solution includes the name of the stream when that stream is not the default data stream (that is, when the stream name is not "::$DATA"). This means that on files with alternate data streams, FESF allows the Client DLL to establish Policy on a per stream basis and not just on a per file basis. Also, while FESF will not call the Policy DLL for directories as a general rule, for file systems that support streams on directories it **will** call the Policy DLL for streams created on directories.

In terms of FESF Policy Caching, caching is done on a per-stream basis. Thus, on file systems that support alternate data streams, the support function *PurgePolicyCacheFile* applies to a specific stream of the file (if the file has multiple data streams).

As a general guideline, any reference in FESF documentation that refers to "files" should be understood to refer to "streams" on file systems that support alternate data streams.

# 3 FESF Components and Interfaces

The basic architectural layout of the FESF system is shown in Figure 1.

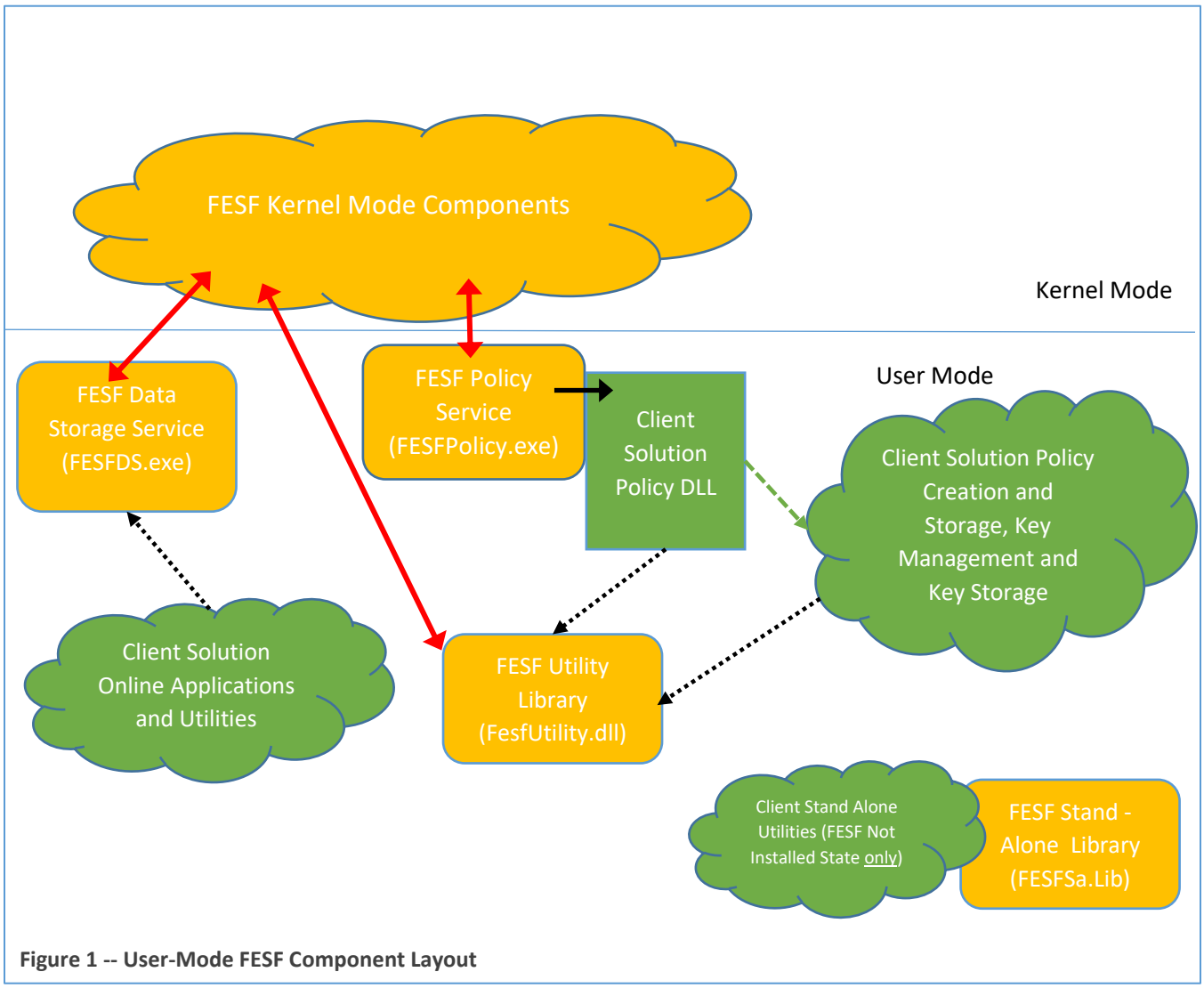


**Figure 1 -- User-Mode FESF Component Layout**

Looking at Figure 1, Components shown in orange are developed, provided, and maintained by OSR. Items shown in green are developed by the Client as part of the Client Solution.

Black solid and dashed lines indicate FESF architecturally defined interfaces that are documented and supported by OSR. The solid black line between the FESF Policy Service and the Client Solution Policy DLL represents a (synchronous) call and return interface. The black dotted lines represent COM interfaces for FESF-provided support and utility functions that may optionally be used by a Client Solution.

The red lines are undocumented, unsupported, interfaces that are private and reserved to OSR and subject to change in future FESF releases. The Client Solution should never invoke the interfaces defined by the red lines directly, but rather should use the public, supported, interfaces designed and provided by FESF.

## 3.1 FESF Kernel Mode Components

The "big orange cloud" in Figure 1 represents the collection of OSR-supplied FESF Kernel Mode Components. These comprise a series of file system mini-filters and their associated libraries. There are a total of four FESF kernel-mode components that are installed as part of FESF: OsrIsolate.sys, OsrDt2.sys, OsrDs2.sys, and OsrSupport.sys.

The FESF Kernel Mode Components are responsible for intercepting file operations (such as **CreateFile**, **ReadFile**, and **WriteFile**) on supported file systems, implementing Client Solution-specified Policies, managing provision of the correct "view" (encrypted/decrypted or raw) of a given file's data based on the Client-specified Policy, and also for performing the actual encryption/decryption operations via Microsoft's CNG kernel-mode library.

Source code for the kernel-mode portions of FESF is not provided as part of the standard FESF kit license.

## 3.2 FESF User Mode Components

The orange rectangles in Figure 1 represent FESF User Mode Components. These components comprise the FesfPolicy service, and two support components: the FesfDs Service and the FesfUtility DLL. OSR reserves the right to extend or add to these components in future FESF releases.

Looking at Figure 1, you'll notice that the FESF Policy Service provides the interface between the FESF Kernel Mode Components and Client Solution components in a system. The support components are provided for use by Client Solution components to make many common operations easier.

### 3.2.1 FESF Policy Service (FesfPolicy.exe)

The FESF Policy Service is the interface between FESF and the components of a Solution that determine Policy. The FESF Policy Service receives requests from the FESF Kernel Mode Components, converts them to the expected format, and passes them to the Client Solution's Policy DLL**.** All calls from the FESF Policy Service into the Policy DLL are done via conventional call/return interfaces.

It is important to understand that the only interface from FESF to a Client Solution is via FesfPolicy, which in turn calls the Client Solution's Policy DLL. While Solution components can call FESF to request information or perform utility functions, all calls that originate from FESF come through FesfPolicy and the Solution's Policy DLL.

The FESF Policy Service calls entry points in the Policy DLL in the context of a worker thread. These calls are synchronous. That is, the Policy DLL must only return when it has the information requested (or else return an error). When a call to the Policy DLL is made, that call is blocking an associated kernel-mode operation. When the Policy DLL returns from a call, the results are returned by the FESF Policy Service to the FESF Kernel Mode Components.

The source code, and all the items necessary for building FesfPolicy from source, are provided as part of your FESF License. This code is for your reference only. OSR does not support changes or customizations to FesfPolicy.

The specifics of the interface between FesfPolicy and the Client Solution Policy DLL, are described later in this document in the section **Policy DLL Callback Function Reference**

### 3.2.2 FESF Data Storage Service (FesfDs.exe)

The FESF Data Storage Service is a Windows service that provides support functions to components of the Client's Solution. The FESF Data Storage Service is an out-of-process COM server that exports the **IFesfDs** interface and a class identifier (CLSID) **FesfDs**.

The functions exported by the FESF Data Storage Service primarily support the manipulation of files with knowledge of the FESF On-Disk Structure (ODS). FesfDs also provides some general utility functions. The services provided by FesfDs include:

- Determining whether a given file is stored in FESF encrypted format.

- Determining the true size on disk of a file stored in FESF encrypted format.
- Retrieving and/or updating the header for a file stored in FESF encrypted format.
- Checking a file stored in FESF encrypted format to determine if it is internally consistent, and optionally attempting to repair that file if problems in the on disk structure are found.

One of the primary features of FesfDs is that it is designed to provide services that can be used in Online, Offline, and FESF Not Installed environments (please be sure to refer to section **2.9 Online, Offline, and Not Installed** for the FESF-specific description of these states). This allows Client Solution components to determine if a given file is encrypted, and even perform limited encryption, decryption, and file validation operations when the FESF Kernel Mode or even User Mode Components are not available to provide assistance. Most of the functionality provided by FesfDs relates to the ODS of FESF encrypted files. This structure defines and controls the in-file storage of the FESF control and consistency information, including the Client Solution's Header Data.

The FESF ODS components are supported in part by common functions or libraries. These support functions are implemented in libraries specifically designed to facilitate multiple OS support (including Windows, iOS, and Linux variants). As of FESF V1.3, FESF provides the libraries necessary to support several types of operations on Windows and Linux systems.

FesfDs currently supports IsFileEncryped, ReadHeader, UpdateHeader and UpdateHeaderWithExtension operations in Online (that is, FESF installed and running) and Offline State (that is, FESF installed but user-mode Solution components are not running). In addition, FESF supports “Stand Alone” versions of certain FesfDS functions that allow applications to perform FESF operations (including encrypting and decrypting FESF files or updating headers on FESF encrypted files) without FESF being installed. This support is provided by the FESF Stand Alone Library, described elsewhere in this document.

The source code, and all the items necessary for building FesfDs from source, are provided as part of your FESF License. This code is for your reference you only. OSR does not support changes or customizations to FesfDs. Also, OSR does not support the direct use of functions that are internal to FesfDS. All operations performed by a Solution on the ODS must be performed through a documented interface. *The only documented interface provided by FESF for ODS operations is through the COM interface provided by the FesfDS Service*. Note that the internal functions called by FesfDS functions will very likely change in future FESF releases.

The list of functions provided by FesfDs are described in detail in the section **FESFDs Function Reference**, later in this document.

### 3.2.3 FESF Utility Library (FesfUtility.dll)

The FESF Utility Library provides support functions to components of the Client’s Solution. The FESF Utility Library is an in-process COM server (that is, a DLL) that exports the **IFesfUtil** and IFesfUtil2 interfaces and a class identifier (CLSID) **FesfUtil.** Because it is implemented as a DLL, the overhead for calling functions in FesfUtility is relatively low.

The FESF Utility Library provides general utility support to Client Solutions, as its name implies. A few of the services that FesfUtility provides are:

- Determining whether the FesfPolicy Service is running.
- Determining whether a given file is stored in FESF encrypted format.
- Determining the true size on disk of a file stored in FESF encrypted format.
- Translating a Volume GUID and file path, as provided by FesfPolicy, to a fully qualified local path specification.
- Retrieving the fully qualified path of a running application, given a Thread ID provided by FesfPolicy.
- Retrieving the Security Identifier (SID) of an executing program, given a Thread ID provided by FesfPolicy.

- Determining if the Security ID (SID) associated with the provided Thread ID is a member of a given Security Group identified by a Security ID.

The source code, and all the items necessary for building the FESF Utility Library from source, are provided as part of your FESF License. This code is provided for your reference only. OSR does not support changes or customizations to FesfUtility.

Additional details about FesfUtility.dll, including documentation for all the functions it makes available, is provided the section entitled **FESFUtility Function Reference** later in this document.

### 3.2.4 FESF Stand-Alone Library (FESFsa.lib)

The FESF Stand-Alone library provides support for implementing Client Solution components that will work in FESF Not Installed state. FESFsa.lib is a cross-platform C/C++ library designed to work on multiple operating systems. Windows and Linux are currently supported.

The FESF Stand-Alone Library provides support for operations that are useful to perform when FESF is in the Not Installed State, such as during a recovery operation or when dealing with FESF encrypted files on a Linux client. Currently, these operations include:

- Decrypting a file that was previously encrypted using FESF.
- Encrypting a file.
- Determining whether a given file is stored in the FESF encrypted format.

The source code for the FESF Stand-Alone Library, and all the items necessary for building FesfSa from source, are provided as part of your FESF License. This code is for your reference you only. OSR does not support changes or customizations to FesfSa. Also, OSR does not support the direct use of functions that are internal to FesfSa.

Additional details about FesfSa.lib, including documentation for all the functions it makes available, is provided the section entitled **FESFSa Function Reference** later in this document.

### 3.2.5 Comparing FesfDs, FesfUtility, and FesfSa

FesfUtility, FesfDs, and FesfSa provide some of the same functions, such as the ability to determine if a file is encrypted. There are a few distinguishing characteristics between the functions provided by FesfUtility, those provided by FesfDs, and those provided by FesfSa. These are summarized in the following table:

| Component | Implemented as… | Available when FESF is in this state |
|---|---|---|
| **FesfDs** | Out of process COM Server | On Line State or Offline State |
| **FesfUtility** | In-Process COM Server | Online State or Offline Line |
| **FesfSa** | Statically linked library | Not Installed State ONLY |

In interpreting the table above, please be sure to refer to section **2.9 Online, Offline, and Not Installed** for the FESF-specific description of the indicated state.

Functions provided by FesfUtility are purely utility-related functions, provided for the convenience of the Client Solution components. Because FesfUtility is an in-process COM server, the overhead of calling functions in this library is minimal (no more than calling a function in any DLL). FesfDs is a Windows service that provides functions as an out-of-process COM server. Calling functions in FesfDs requires marshaling arguments, sending them to a separate process, and context switching into a thread in that process to service the request. Thus, the overhead of calling a function in FesfUtility is significantly lower than calling a function in FesfDs.

Functions provided by FesfSa are only for use in FESF Not Installed State. This is a critical restriction that must not be taken lightly. Using utilities that build with FesfSa on systems where FESF is running will result in unpredictable results, including potential loss of data.

## 3.3 Client Solution Components

The Client Solution will comprise as few or as many components as required to implements its design goals. Components of the Solution may be local to or remote from any given system or (most likely) a combination of the two. The only part of the Client Solution that is required by FESF is a Policy DLL that will be called by the FESF Policy service.

### 3.3.1 Client Solution Policy DLL

The Client Solution Policy DLL is provided by the Client. OSR includes a complete and well-documented sample Policy DLL (SampPolicy) that Clients can use as the basis for their own implementation. See the **FESF Sample Solution Guide** for more information on the OSR-provided sample code.

As previously described, the Client Solution's Policy DLL is the primary interface point between FESF and the Client's product implementation. Except for the initialization callback which is always called by name, callback functions in the Policy DLL are called by pointer. The Policy DLL passes pointers to each of its callback functions during initialization processing. After initialization, FESF calls callback functions in the Policy DLL to determine policy for a particular open instance of a file, as well as to retrieve the Policy DLL defined Header Data and Key data for files that are to be encrypted/decrypted by FESF.

As an example of how things work, consider the Policy DLL's *PolGetPolicyNewFile* function. This function is called whenever a new file is being created on a supported file system. After the **CreateFile** has been successfully processed by the target file system but before the user's call to open the file has completed, the FESF Policy Service calls the Policy DLL's *PolGetPolicyNewFile* callback function to determine if data subsequently written to this file should be encrypted. If *PolGetPolicyNewFIle* indicates that the file is to be encrypted, FESF calls the Policy DLL's *PolGetKeyNewFile* to retrieve the Algorithm ID, Key, and Policy DLL specific Header Data to be stored with the file to allow the file to be decrypted at a later time. During the call into the Policy DLL, the user application that called **CreateFile** (and the kernel-mode mechanism associated with this operation) is blocked, waiting, until both *PolGetPolicyNewFile* and *PolGetKeyNewFIle* return. As a result, all processing done in the Policy DLL must be prompt.

Processing for other callbacks in the Solution's Policy DLL work similarly. The calls to *PolGetPolicyExistingFIle*, and *PolGetKeyFromHeader* take place after the application's **CreateFile** operation has been processed by the file system on which the file is located, but before the user is informed of the result. Again, this call into the Client's Policy DLL is blocking completion of Windows' kernel mode processing of this open operation and ultimately the application's further progress.

#### *Note on lack of serialization among* Policy DLL *callbacks*

A note is probably appropriate here about parallel operations. The FESF system as a whole is intrinsically asynchronous. This reflects the way that the Windows OS does its work, and is also considered "best practice" in terms of overall system performance and throughput. As a result of this asynchronous design, multiple calls to the Policy DLL can take place in parallel. FESF provides no serialization for calls into the Policy DLL**.** Thus, it will be typical for multiple threads to call into the Policy DLL simultaneously. In fact, it is even possible for the Policy DLL to get multiple simultaneous calls to the same callback function, such as *PolGetPolicyNewFile* for the same file. This is possible if two threads attempt to create the file at the same time.

In the case that two different threads both simultaneously attempt to create the same file, only one of them will ultimately succeed (in kernel-mode processing). FESF will ignore the result returned by the Policy DLL from the second

(and, hence, unsuccessful) attempt. Subsequent attempts to retrieve the Key Data and Header Data will consistently return the same Key Data and Header Data that was actually used by the file.

This can get even more confusing, however, when multiple threads attempt to access an existing encrypted file simultaneously. This could result in multiple simultaneous calls to the Policy DLL's *PolGetPolicyExistingFile* callback. If the openers each provide appropriate share access, multiple openers can succeed. So, in this case, the results returned by the Policy DLL's *PolGetPolicyExistingFile* callback will all be relevant.

In summary, when the FESF Policy Service calls the Policy DLL, that call is synchronous in that an associated file operation is being blocked while this call is in progress and the operation will only continue when the Policy DLL returns. However, the FESF Policy Service will call the Policy DLL's entry points in parallel from multiple worker threads (perhaps a few hundred!). The same callback function in the Policy DLL can be called an almost limitless number of times in parallel, and multiple different functions in the Policy DLL can also be called in parallel. It is the job of the Policy DLL (and any user-mode components with which it communicates) to provide whatever serialization may be required.

# 4 Designing and Building a Solution

As previously described, the design of a given Client Solution is dependent almost entirely on the design goals and scope of that Solution. The only required component of any given Client Solution is the Policy DLL. In this section, we'll describe the interface functions and major points to consider in implementing a Policy DLL.

The FESF Policy Service calls callback functions in the Policy DLL to determine Policy, gather data, provide an opportunity for the Policy DLL to exercise control over particular functions, and to perform other support operations. The Policy DLL is loaded dynamically by the FESF Policy Service via a call to the Windows function **LoadLibrary** based upon the FESF configuration information in the Windows Registry (see the Section entitled Arranging for FESF to Load Your Policy DLL). After the Policy DLL is loaded, FESF calls **PolicyDllInit** to allow the Policy DLL to perform initialization processing. This initialization includes the Policy DLL calling **FePolSetConfiguration** to inform FESF of various configuration choices, including providing pointers to FESF for the callback functions that the Policy DLL supports.

While the role of the Client Solution's Policy DLL is always the same in any FESF system (it is always the primary interface between FESF and the Client's implementation), different Client Solution architectures may result in the Solution's Policy DLL doing very different processing. In some architectures, almost no processing is done in the Policy DLL aside from argument preparation and data marshalling. The Policy DLL is essentially stateless. In these architectures, actual policy determination and key management is done by a service with which the Policy DLL communicates. That service may either be hosted locally (on the same system as the Policy DLL) or remotely (on a server on a LAN system, for example).

In other Solution architectures, Clients may design their Policy DLL to be a more active participant in policy determination. In these architectures, the Policy DLL might store policy and key information locally, and only invoke a remote policy and/or key management service when local information is not available.

And, of course, there are infinite variations on the two Solution architectures that we've described.

Each approach to building a Policy DLL has its particular advantages and disadvantages. The Solution chosen will ultimately depend on what best meets the needs for the overall product being built and the environment in which it will be used. In any case, FESF does not impose any specific requirement or restriction on the Client Solution architecture, beyond the requirement that returns from Policy DLL callback functions must be prompt.

As an example of one basic approach to building a Solution using FESF – and as a demonstration of how to use the provided support services and perform common operations – OSR provides the complete source and executables for a Sample Solution. For more information about this sample, please refer to the **FESF Sample Solution Guide**.

## 4.1 Policy DLL Callback Functions

The possible callback functions that a Policy DLL can support are:

- **PolicyDllInit** – This is the only entry point that is called by name, and it must be named **PolicyDllInit**. This callback function is called by FESF immediately after the Policy DLL has been loaded to allow the Policy DLL to perform initialization processing. This processing must include initializing a **FE_POLICY_CONFIG** structure and filling it in with pointers to the other callback functions supported by the Policy DLL. The **FE_POLICY_CONFIG** must then be passed to FESF by the Policy DLL calling **FePolSetConfiguration** during its **PolicyDllInit** callback function processing. This callback function is required, and must be implemented by every Policy DLL.

- *PolGetPolicyNewFile* – Called when a new file is being created with write access requested on a supported file system, to determine if the file should be created in FESF encrypted format. This callback function is required, and must be implemented by every Policy DLL. Note that for the purposes of FESF, a "new file being created"

include an existing zero-length file being opened or the destructive create of any existing file. See the reference pages for *PolGetPolicyNewFile* for the details

- *PolGetKeyNewFile* – Called when a new file is being created in FESF encrypted format to get the Key Data and Algorithm ID to be used to encrypt data for the file, and the Policy DLL Header Data that will be stored with the file. This callback function is required, and must be implemented by every Policy DLL.

- *PolGetPolicyExistingFile* – Called when an existing FESF encrypted file is being opened to determine if encrypted data read from the file should be decrypted before it's returned and whether data written to the file should be encrypted before it's written. This callback function is required, and must be implemented by every Policy DLL.

- PolGetPolicyDirectoryListing – Called when a directory is opened to determine whether the sizes returned in a directory listing will reflect what is consumed on disk (allowing for the Solution Header) or just the size of the data in the file.

- *PolGetKeyFromHeader* – Called when an existing FESF encrypted file is being opened, and the Policy DLL has previously determined that the opening handle will receive transparent encrypted/decrypted access. Given the Thread ID of the thread performing the access and the Policy DLL Header Data that FESF stored with the file, the Policy DLL returns the Key Data and Algorithm ID to be used for encryption and decryption operations. This callback function is required, and must be implemented by every Policy DLL.

- *PolApproveRename* – Called when a file on a supported file system is being renamed. The Policy DLL can choose to allow or disallow the operation for security purposes. This callback function is optional.

- *PolApproveCreateLink* – Called when a hard link is being created on a supported file system. The Policy DLL can choose to allow or disallow the operation for security purposes. This callback function is optional.

- *PolApproveTransactedOpen* – Called when a transactional open is encountered. All transactional opens are treated by FESF as raw. This call gives the Solution the option of disallowing the open completely. A Solution might choose to do this if allowing the open could lead to secure data leakage. This callback function is optional.

- *PolReportFileInconsistent* – Called when FESF discovers a file that is in FESF format, but that has an internal inconsistency or format error. This callback function is optional.

- *PolFreeHeader* – Called by FESF to return the storage for Policy DLL Header Data that was previously allocated by the Policy DLL. This callback function is required, and must be implemented by every Policy DLL.

- *PolFreeKey* – Called by FESF to return the storage for the Key Data that was previously allocated by the Policy DLL. This callback function is required, and must be implemented by every Policy DLL.

- *PolReportLastHandleClosed* – Called when the last handle to a file in FESF encrypted format is being closed. This callback function is optional.

- *PolDllUninit* – Called during shutdown to allow the Policy DLL to perform any cleanup operations it requires.

Aside from the required functions, a Policy DLL only needs to implement those functions that are relevant to the Client product.

## 4.2 Policy DLL Initialization

Every Policy DLL must implement a **PolicyDllInit** callback function. This is the only Policy DLL callback function that is called by name.

The purpose of the **PolicyDllInit** callback function is to perform Policy DLL initialization. This initialization must include calling **FePolSetPolicyConfiguration** to select configuration options and provide FESF pointers to the other callback functions that the Policy DLL supports.

To be able to call **FePolSetPolicyConfiguration**, the Policy DLL builds an **FE_POLICY_CONFIG** structure. This structure is typically allocated on the stack by the Policy DLL. The structure must be zeroed before use.

Also specified in the **FE_POLICY_CONFIG** structure are a list of encryption algorithms and options, and a unique handle that will be used by the Policy DLL to identify each specific algorithm/option pair.

The **FE_POLICY_CONFIG** structure is also the place where the Policy DLL indicates how FESF should handle rename and hard link operations when it is in Offline State. The Policy DLL also indicates how FESF encrypted files that are internally inconsistent should be treated in Offline State.

## 4.3 Returning Failure from Policy DLL Callbacks

The Policy DLL callbacks *PolGetPolicyNewFile, PolGetKeyNewFile, PolGetPolicyExistingFIle, and PolGetKeyFromHeader* can all return failure indications. Returning failure from these functions should be avoided, if possible, and reserved only for serious error conditions.

The reason for this recommendation is that these callbacks are called by FESF after the user has successfully opened the given file. When the Policy DLL returns an error, the user will receive an error back from what was otherwise a successful create operation. When that open operation includes a "destructive create" (an open operation that supersedes or overwrites an existing file) the contents of the existing file have already been deleted. If the open operation results in a new file being created, that new file has already been created on disk when the Policy DLL's callback is called.

In these cases, returning an error from one of the previous mentioned functions can result in an empty file being created on the system. FESF does not attempt to clean-up these empty files in any way.

## 4.4 Guidance for Implementing Callback Functions

Regardless of the design of your Solution, there are four important things we'd like you to keep in mind in terms of the design and implementation of Client Policy DLL callbacks. Those four things, in no particular order, are:

- **All Policy DLL callback code that you implement must be thread-safe**. FesfPolicy dynamically grows (and shrinks) the pool of worker threads it uses to call Policy DLL callbacks. FESF currently sets the maximum number of active callback threads to 122 on 32-bit systems, and to 244 on 64-bit systems. These maxima are subject to change, up or down, in subsequent releases of FESF. During pre-release testing, we regularly hit the maximum number of active worker threads. A design which takes maximum advantage of the parallelism offered by FESF, and an efficient, scalable, locking scheme where access to shared data is required, are a must in your Solution.

- **Callbacks to your policy DLL must complete "promptly."** Remember, when FesfPolicy calls your Policy DLL it's blocking a kernel-mode operation, typically a user's request to open or create a file. For the Solution and the overall Windows system on which the Solution is running to exhibit good performance, prompt and efficient processing is a must. A Solution architecture that judiciously caches information, including Policy decisions, key material, and user Security Group membership, is strongly advised. While we're not fans of premature optimization, we would encourage you to at least take these precepts into account as part of your Solution's initial design.

  You might reasonably ask "What time period, precisely, does 'promptly' imply?" Unfortunately, we don't have a good answer for you. By promptly, we really mean "as soon as practically possible for your Solution." Each workload will be different, and your Solution has to meet your design and usability goals. However, from a systems perspective, we would advise targeting a small number of seconds (in the low single digits) as the maximum time for a Policy DLL callback to complete under heavy system load. This should yield acceptable performance. This is provided only as a guideline to aid you in your design.

  One absolute maximum that we can warn you about is that used by the FESF Kernel Mode Components. These components set an arbitrary maximum of 30 seconds that they will wait for a reply from user-mode. One of our developers describes this interval as "a virtual eternity", and indeed it is the uppermost bound that can be expected for a reply even on a severely degraded system. After this period of time, an error message is logged to the Windows Event Log and the timed-out operation is completed with an error (access denied). While this will ensure the system remains running, returning errors from a Policy DLL callback is rarely desirable (see the next item below).

- **Avoid returning failure codes** when possible. As described in the section above entitled **Returning Failure from Policy DLL Callbacks**, it is almost always a bad idea to return a failure code from the *PolGetPolicyNewFile, PolGetKeyNewFile, PolGetPolicyExistingFIle, and PolGetKeyFromHeader* Policy DLL callbacks. We would advise you to restrict failure returns to those conditions which are unforeseeable and catastrophic. We would recommend not returning failure statuses to FESF callbacks in transient conditions such as lost connections or slow responses from your key management server. Obviously, only you understand your Solution and its requirements. But, at the very least, please do not return an error from these callbacks as an alternative form of implementing file access security.

- **Be conscious of the potential for reentrancy,** and avoid it. Bear in mind that your Policy DLL's callbacks are executing while a Windows system service (typically a CreateFile operation) is pending. This means you must avoid the potential for reentrancy problems. As a very simple example, consider what might happen if you try to create a new file each time you're called in you *PolGetPolicyNewFile* callback. In this example, your *PolGetPolicyNewFile* callback would get called-back endlessly (each time, creating a new file which then results in another callback). Not good!

## 4.5 Guidance Regarding Policy DLL Header Data

When a Solution's Policy DLL determines that a newly created file should be encrypted, FESF subsequently calls the Policy DLL's *PolGetKeyNewFile.* In response to this callback the Policy DLL returns encryption key information, as well as an initial copy of Policy DLL defined Header Data for FESF to store with the newly created file.

The Policy DLL defined Header Data may contain any data the Solution may require to derive the encryption key information for the file on subsequent accesses. However, FESF will attempt to significantly optimize storage of

Header Data that is “small.” FESF currently defines “small” as being less than 200 bytes in length. This value is subject to change in subsequent FESF releases, as is the manner and extent that FESF choses to optimize storage for small files.

The only supported method for a Solution to retrieve or update the Header Data stored with an FESF encrypted file is by using documented functions, such as those supplied by FesfDS (while FESF is installed) or FesfSa (when FESF is not installed on the system) and described later in this document.

## 4.6 Working with Local, Network, and Shadow Volume File Paths

File paths are expressed to the Policy DLL via an **FE_POLICY_PATH_INFORMATION** structure. This structure is used to describe both local file paths as well as network file paths.

In the case of a local file path, the volume GUID of the local volume is supplied along with the volume relative path to the file. The Policy DLL may use the volume GUID to look up a “friendly name” for the volume, such as a drive letter or mount points, using standard Windows APIs. The Policy DLL may also call the FESF Utility function GetFullyQualifiedLocalPath, which will attempt to convert the volume GUID into a friendly name and concatenate the supplied relative path. You can read more about volume GUIDs, drive letters, and mount points by consulting the MSDN documentation.

If the volume GUID supplied is **FE_NETWORK_GUID**, the path supplied represents a file located on a network share. Network shares do not have volume GUIDs and thus this GUID is meaningful only within the bounds of the FESF Policy DLL interface and as input to certain FesfDs APIs. The server and share are supplied along with the share relative path to the file.

Network paths have some potentially unexpected behaviors that warrant additional discussion. Of particular note is the fact that FESF components make no attempt to normalize or rationalize the server component of the UNC path. For example, let’s assume that there is a server EmployeeFiles in the FESFTest domain with two IP addresses 10.0.0.10 and 10.0.0.11. A user might access a file on this server using any one of these paths:

- \\EmployeeFiles\Records\Doc.docx
- \\EmployeeFiles.FESFTest.com\Records\Doc.docx
- \\10.0.0.10\Records\Doc.docx
- \\10.0.0.11\Records\Doc.docx

In each of these cases, the Policy DLL will see the server name *as specified by the requestor*. Thus, if the Policy DLL is using a strict path based policy these paths may appear to represent different files located on different servers. If the Policy DLL requires a unique canonical name to represent the server, the Policy DLL must extract the server component of the supplied server and share information and use an external source to resolve the name.

Another interesting case is when the user accesses a network share via a drive letter mapping. For example, a user may map her Z: drive to the \\EmployeeFiles\Records share. If the user then accesses Z:\Doc.docx, the Policy DLL *will not see the drive letter based path* in its callbacks. As stated previously, FESF always passes the Policy DLL a standard UNC path, even if the file access was made via a drive letter mapping. Note that the server component of the path is still subject to the ambiguity specified above, but in this case it is dependent on the server name format used when the drive letter mapping was created.

Finally, please note that like network paths, FESF does not attempt to do normalization of hard-links. Thus, a given file can have numerous hard-links that point to it. Unless you’re aware of this, you can may at times get unexpected results, including as result values from a function such as GetExecutablePathForThreadId.

If the volume GUID supplied is **FE_SHADOW_VOLUME_GUID** , the path supplied represents a file located on a local shadow volume. Shadow volumes do not have volume GUIDs and thus this GUID is meaningful only within the bounds of the FESF Policy DLL interface and as input to certain FesfDs APIs. Paths to files on shadow volumes carry the shadow volume "device name". This can be used as input to find out more about the shadow volume. For instance, the VSS APIs provide such support. A file on a shadow volume can be opened by constructing a UNC file name by appending the file name to the shadow volume device name and prepending the while with \\?\GlobalRoot\

Thus:

\\?\GlobalRoot\device\HarddiskVolumeShadowCopy9\dir\file

## 4.7 A Note About Raw File Access

As previously described, FESF encrypted files are stored using an On Disk Structure (ODS) that is proprietary to FESF and subject to change in subsequent FESF releases. Remember that the only supported mechanism for retrieving and updating the Policy DLL supplied Header Data is via documented FESF functions. It is an architectural violation for a Solution Component to bypass the supported FESF functions and instead use raw access to manually update any portion of the FESF ODS, including the Header Data.

Finally, because when we talk about encrypting files we're usually dealing with security, it's useful to keep in mind the fact that any application that has write access to a file can change all or part of the that file's contents. This applies equally to FESF encrypted files as it does to ordinary unencrypted files. When an application is given raw *write* access to an FESF encrypted file, that application could potentially overwrite or otherwise damage the file. Again, this is no different than any file on Windows that relies on a specific file format (whether that's an executable image, a database file, or a Word document).

## 4.8 Arranging for FesfPolicy to Load Your Policy DLL

The FESF Policy Service loads the Policy DLL dynamically during system startup, using the standard Windows **LoadLibrary** function. The Policy DLL is loaded based on the file specification store in the Registry under the following path:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FESFPolicy\Parameters

**Value Name:** PolicyDll
**Value Type:** REG_SZ
**Value:** <String specifying path to the Client's Policy DLL>

In specifying the value, the standard rules for how **LoadLibrary** interprets the provided string apply. See the documentation for the *lpFileName* parameter for **LoadLibrary** in MSDN for more information.

There are two things of which to take note when changing the Policy DLL:

1. You must stop and restart the FESF Policy Service (FESFPolicy.exe) for the new Policy DLL to be loaded. The FESF Policy Service only reads the PolicyDll Registry value during initialization.

2. Be aware that any files that were encrypted by FESF by previous versions of the Policy DLL will be recognized as encrypted files by FESF. Your Policy DLL may, therefore, be called at *PolGetKeyFromHeader* with Header Data that was created by a previous Policy DLL. It is therefore wise to include some identifying information in the Policy DLL Header Data so that your Policy DLL can validate and recognize the header before using its contents.

## 4.9 FESF Policy Service and Kernel Component Logging

To aid in debugging Policy DLL implementations, and also to assist with diagnosing problems with the encryption subsystem in the field, FESF includes logging facilities.

### 4.9.1 FesfPolicy Logging

FesfPolicy logs messages to the Windows Event Log. Messages are logged for all unusual and anomalous events, including both fatal and non-fatal issues. Messages are logged to the Windows Logs\Application\FesfPolicy log.

The FESF Policy Service includes the capability to log messages to the console whenever errors are encountered in its interactions with the Policy DLL.

To make debugging and testing easier, the FESF Policy Service can be run interactively from the command line (instead of being installed and run as a Windows service). When you start FesfPolicy interactively, the command line to use is:

FesfPolicy [-debug | -DEBUG] [logfile]

Starting FesfPolicy interactively and specifying one of the debug switches, will cause the Policy Service to output debug messages to the stderr (which is the command prompt window or the Visual Studio Output window if FesfPolicy is run from within Visual Studio).

The two forms of the "debug" switch are equivalent, except that when the uppercase version is used (that is "-DEBUG") FesfPolicy will issue a debug break point during startup.

When either of the "debug" switches are present, output can optionally be redirected to a file instead of to the console.

There is also a –SDEBUG option that, when FesfPolicy is run as a service, causes the service to conveniently issue a breakpoint during startup.

The FESF Policy Service may optionally be compiled to output debug data using **OutputDebugString**. See the source code for more information.

### 4.9.2 FESF Kernel Component Logging

The FESF Kernel Mode Components log basic status messages, including errors, to the Windows System Event Log. For example, when FESF starts, FesfDt2 logs the message shown in Figure 2.

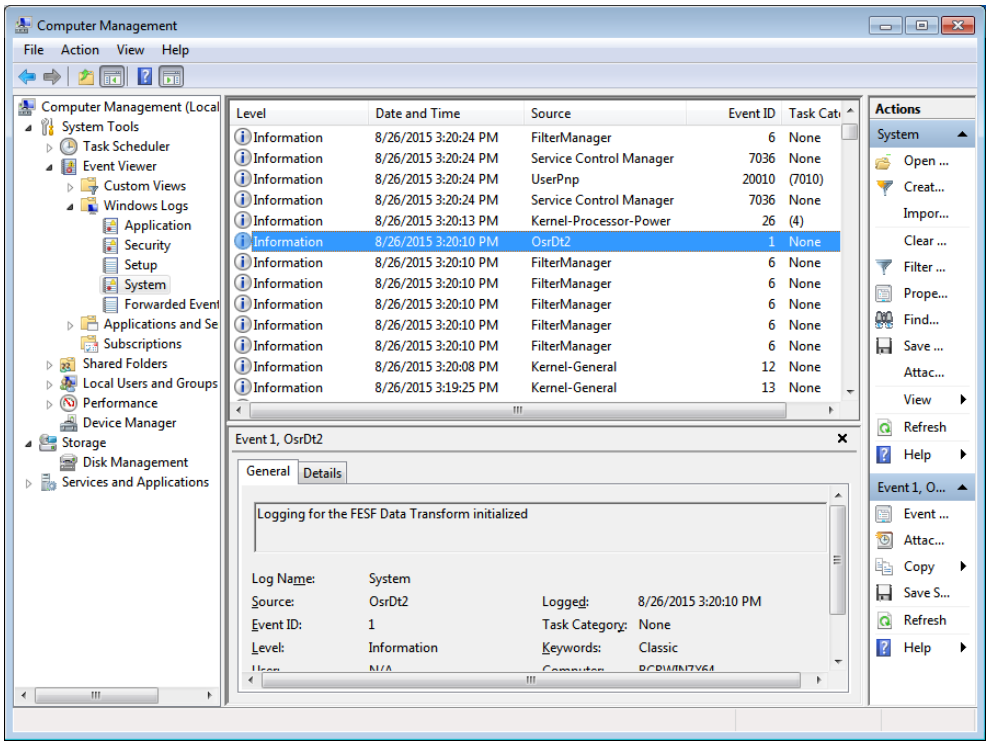


**Figure 2 -- FESF Kernel Mode Component Logging**

# 5 Building FESF Sources

## 5.1 Building the Provided Source Code (Windows)

The FESF Software Developer's Kit contains two Visual Studio 2013 solutions, along with the necessary header and library files to successfully build them. As provided, the solutions build from source without errors or warnings (at /W4) and pass Visual Studio Code Analysis at the Microsoft Native Recommended Rules level. Code analysis is enabled for every build so expect build times to be longer than you might otherwise expect.

To build these solutions:

1. Copy the \Src directory from the archive to your development system, where Visual Studio 2013 is installed.
2. Open the FESF user mode solution, \src\UM_FESF\UM_FESF.sln, in Visual Studio and build for the desired configuration.
3. After building the FESF user-mode solution, build the sample solution, \src\UM_Sample\UM_Sample.sln, in Visual Studio 2013.

The built user-mode components will be placed in the platform specific directories of the solution tree: debug, release, x64\debug and x64\release.

## 5.2 Building the Provided Stand Alone Utility Source Code on Linux

The FESF Software Developer's Kit also contains two makefiles that can be used to build the Stand-Alone library and the sample Stand-Alone utilities.

***The Stand-Alone sample utilities are provided to illustrate the use of the FesfSa library. Please note that the provided utilities utilize a demonstration encryption facility that is not compatible with the FESF Sample Solution.***

***To build Stand-Alone utilities that are compatible with the FESF Sample Solution, you will need to implement encryption and decryption algorithms that implement AES 256 CBC mode, with ESSIV, and 256 byte blocks.***

To build these the Stand-Alone Library and Sample Stand-Alone Utilities:

1. Copy the /Src directory from the archive to your Linux development system.
2. Download the Windows Development Kit on a Windows system, and copy no_sal2.h to an include directory on your Linux development system.
3. Run make on the Stand-Alone library, from the /src/UM_FESF/FESFSa/ directory
4. Run make on the Stand-Alone encryption/decryption sample, from the /Src/UM_Sample/SampSaEncDec directory.

The build user-mode components will be placed in the same directory as the makefile.

## 5.3 Building and Installing Your Own Solution

The single most important thing you can do to help ensure the success of your Solution development project is for development team members to thoroughly read and understand the **FESF Solution Developer's Guide** (provided in the Docs directory of the FESF Software Developer's Kit). This guide should tell you most of what you need to be able to successfully design and develop a product using FESF.

Looking at the FESF Sample Solution will illustrate where the various header (.h) and type library (.tlb) files are located in the FESF development kit.

While the structure of your source tree is certainly up to you, we recommend locating your Client Solution directory as a peer (that is, on the same directory level) of the UM_FESF directory. We think this will make reference to the FESF components easier.

### 5.3.1 Supported Toolset

The FESF V1.5 Release supports Visual Studio 2017. The Stand-Alone linux components are supported by versions of g++ that have C++11 support.

# 6 Notes on Installing FESF with Your Solution

When it comes time to creating a kit for your Client Solution, you will likely create a procedure that installs both your Solution components and the FESF components on a target system. The steps required to install FESF include the following:

- Copy the appropriate kit directory to the test system where your solution will be installed. Let's assume you copy it to "C:\Program Files\Osr\FESF V1.5".

- Install the FESF kernel-mode drivers: There are 4 drivers that comprise the kernel mode portion of FESF – osrsupport.sys, osrisolate.sys, osrds2.sys and osrdt2.sys. The drivers can be installed via their associated INF files. As of the FESF V1.3 Release, the release versions of the FESF kernel mode components in the kit directory are signed by OSR and attestation signed by Microsoft.

  Note that after installing each of the drivers via their associated INF files, the installation will ask you if you want to reboot now or later. Select later and complete all installation steps before rebooting.

  One potential method you may choose to install the drivers from your custom installation application (that properly uses the INF files) is to use DIFxApp – the Driver Install Frameworks for Applications. There are, of course, other equally good options. However, we do urge you to be sure that whatever option you choose does use the provided INF files, as recommended by Microsoft.

- Install the FESF user mode component FESFPolicy.exe: This service can be installed from an elevated command prompt using Microsoft Windows tool sc.exe with the following commands:

```
C:\> sc create FESFPolicy type= own start= auto error= normal binPath=
"C:\Program Files\Osr\FESF V1.5\FESFPolicy.exe" depend=
OSRIsolate/OsrDs2/OsrDt2 DisplayName= "OSR FESF Policy Service"
```

  Note that the equal sign ("=") is part of the option name and that a space is required between the equal sign and the value.

  Depending on the technology you choose for your custom installer, set up the installation of this user-mode service based on the parameters shown in the above command line.

- Install the FESF user mode component FESFDs.exe: This service can be installed from an elevated command prompt, as follows:

```
C:\> C:\Program Files\Osr\FESF V1.5\FESFDs.exe /Service
```

  Note that the first time this service is run it will be self-registered as an out-of-process COM server.

  Depending on the custom installation technology chosen, set up the installation of this service based on the information supplied here.

- Register the FESF utility DLL FESFUtility.dll as an in-process COM server: This dynamic link library can be registered using the Microsoft Windows tool regsvr32.exe:

```
C:\> Regsvr32 FesfUtility.dll
```

  Depending on the installation technology chosen, set up the registration of this DLL based on the information supplied here.

- Create a Registry value of type REG_SZ under the path:

  `HKLM\SYSTEM\CurrentControlSet\Services\FESFPolicy\Parameters\PolicyDLL`

  that contains a path to your own policy DLL.

- Install/set up any additional components that are required for your solution.

### 6.1.1 How to Rename your Drivers

We do not recommend changing the names of the OSR-supplied kernel-mode drivers in FESF V1.5. We have performed very limited testing with drivers using other than the default names. Still, we understand that some Clients may desire to customize the name of their kernel-mode components. With the warning above, we provide this guidance for those Clients.

If you decide to rename the drivers or their Filter Manager Instances, you will need to make the following configuration changes:

- Provide FESFPolicy.exe with the name of the DT Driver (default is OsrDt2) by setting the following REG_SZ value to the driver name:

  `HKLM\SYSTEM\CurrentControlSet\Services\FESFPolicy\Parameters\DtDriverName`

  NOTE: Ensure that this is correct! Setting the wrong value will apparently work, but your configuration information will not be communicated to the Driver. You can test this (after starting the service) by checking to see whether the key `HKLM\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Algorithms` has subsidiary keys.

- Provide the Dt driver with the name of the Isolate and the Ds drivers (defaults are OsrIsolate and OsrDs2, respectively) by setting the following REG_SZ values to the driver names:

  `HKLM\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Parameters\IsolateDriverName`

  `HKLM\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Parameters\DsDriverName`

- If you change the name of the Instances that the drivers attach as (not recommended), this information must be provided to the Dt driver by setting the following REG_SZ values under the key:

  `HKLM\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Parameters`

  - IsolateInstance – The identifier for the Isolate driver Instance (default is “Isolate Instance”)
  - DsInstance – The identifier for the Ds driver instance (default is “Ds Instance”)
  - UpperInstance – The identifier for the Dt’s upper instance (default is “DtUpperInstance”)
  - LowerInstance – The identifier for the Dt’s lower instance (default is “DtLowerInstance”)

# Policy DLL Callback Function Reference

The functions described in this section are prototypes for callbacks that may be implemented by the Client Solution's Policy DLL. Some of these callbacks are required, others are optional. The status of each function is noted in that function's description.

# About Implementing Your Callback DLL

## Regarding SAL Annotations

If you look at PolDllApi.h, where the FESF Policy DLL structures and callback functions are all defined, you'll almost certainly notice that the function prototypes for the callback functions are filled with what might look to you to be strange notes. For example:

```
_Success_(return == true)
FESFAPI
POL_GET_KEY_NEW_FILE(
    _In_ FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_ DWORD ThreadId,
    _Outptr_result_bytebuffer_(*PolHeaderDataSize) PVOID *PolHeaderData,
    _Out_ _Deref_out_range_(>, 0) DWORD *PolHeaderDataSize,
    _Outptr_result_z_ LPCWSTR *PolUniqueAlgorithmId,
    _Outptr_result_bytebuffer_(*PolKeySize) PVOID *PolKey,
    _Out_ _Deref_out_range_(>, 0) DWORD *PolKeySize,
    _Outptr_result_maybenull_ PVOID *PolCleanupInfo
    );
```

All the weird stuff that doesn't look like C++ are "annotations" in Microsoft source-code annotation language (SAL). These annotations describe to the compiler and to Visual Studio Code Analysis how the various parameters of a function are to be used. We're big fans of SAL Annotations here at OSR because they've helped us find and prevent more bugs to date than we could ever count. You can read a bit about SAL Annotations in our blog post: http://www.osr.com/blog/2015/02/23/sal-annotations-dont-hate-im-beautiful/

How do these annotations affect you? Well, mostly, they won't. Except that if you enable Visual Studio Code Analysis you'll be able to find problems with how you're implementing your Policy DLL callbacks more quickly.

There is one interesting issue that you'll encounter during implementation, however. And that issue is that you will almost certainly not want to duplicate all the SAL Annotations when you declare or define your callback function. In other words, when you write the code that implements POL_GET_KEY_NEW_FILE, you probably won't want to have to include the SAL for each of your parameters. And, fortunately, that's easy to avoid.

When declaring a callback function in your Policy DLL's header file, we recommend that you use the type definition that we provide in the first line of the Syntax section of the documentation. The location of the type definition is shown in Figure 3.

In your implementation file, when you define the code for your callback function, we suggest that you use the single annotation **_Use_decl_annotations_**. This will avoid you having to duplicate the annotations, and leave your source code clean and easy to read.

We've provided an example of this approach below.

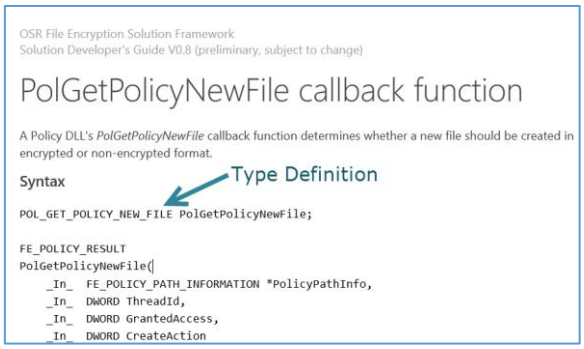


**Figure 3 -- Type Definition**

If you're implementing the *PolGetKeyNewFile* callback, you'd declare your function in your header file like this:

```
POL_GET_KEY_NEW_FILE     MyPolicyDllGetNewFileKey;
```

And in your implementation file, where you define your function, your code would look like this (note the use of _Use_decl_annotations_ has been highlighted):

```
_Use_decl_annotations_
bool MyPolicyDllGetNewFileKey(
    FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    DWORD ThreadId,
    PVOID *PolHeaderData,
    DWORD *PolHeaderDataSize,
    LPCWSTR *PolUniqueAlgorithmId,
    PVOID *PolKey,
    DWORD *PolKeySize,
    PVOID *PolCleanupInfo)
{
    //
    // Get the fully qualified executable path
    //
    CComBSTR exePath;
    HRESULT hr = g_pFesfUtil->GetExecutablePathForThreadId(ThreadId, &exePath);

    if (FAILED(hr))
    {
        return false;
    }
  // ... and the rest of your function goes here...
```

Combining the use of the Type Definition and _Use_decl_annotations_ will get you all the benefits of using SAL Annotations for your callbacks, without having to look at any of the ugly annotation text.

# PolicyDllInit callback function

A Policy DLL's **PolicyDllInit** callback function is the first Policy DLL function called by FESF. A Solution's Policy DLL is responsible for performing initialization in this function.

## Syntax

```
bool
PolicyDllInit(
    VOID
)
{ ... }
```

## Parameters

(none)

## Return value

If **PolicyDllInit** succeeds, it returns (the C++ bool value) **true**. Otherwise, it returns **false**.

Returning false will result in FESF not calling any further functions in the Policy DLL.

## Remarks

All Policy DLLs must implement this callback function. If this callback function is not implemented, FESF will run in Offline State. This callback must be named **PolicyDllInit**, and is the only Policy DLL callback that the FESF Policy Service locates by name.

A Policy DLL's **PolicyDllInit** callback function is called by FESF to allow the Policy DLL to perform initialization. The Policy DLL first performs any internal initialization it may require and then must call **FePolSetConfiguration**.

In terms of FESF, the primary operation performed by the Policy DLL within **PolicyDllInit** is to build an **FE_POLICY_CONFIG** structure and pass a pointer to this structure to FESF by calling **FePolSetConfiguration**. The **FE_POLICY_CONFIG** structure contains the following information:

- The version of the FE Policy Interface supported by the Policy DLL.
- Whether FESF Policy Caching should be enabled.
- Default offline behaviors for hard link creation, rename operations, and inconsistent file handling.
- A list of one or more CNG Algorithms and associated properties, along with a unique algorithm identifier (the Algorithm ID) that will be used by the Policy DLL to refer to each.
- Pointers to the Policy callback functions that are implemented by the Policy DLL.

As soon as **FePolSetConfiguration** has been called, FESF will begin calling callbacks in the Policy DLL.

To enable clean-up operations FESF will call the Policy DLL's *PolUnInit* callback function during shutdown. Note that *PolUnInit* is called through the pointer provided in the FE_POLICY_CONFIG structure and is not located by name.

## Examples

For an example implementation of **PolicyDllInit**, see the documentation for **FePolSetConfiguration.**

## See also

The FESF Sample Solution contains an example implementation of this callback function.  This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolGetPolicyNewFile callback function

A Policy DLL's *PolGetPolicyNewFile* callback function determines whether a new file should be created in encrypted or non-encrypted format.

## Syntax

```
POL_GET_POLICY_NEW_FILE PolGetPolicyNewFile;

FE_POLICY_RESULT
PolGetPolicyNewFile(
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_  DWORD ThreadId,
    _In_  DWORD GrantedAccess,
    _In_  DWORD CreateAction
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]
A pointer to an FESF allocated FE_POLICY_PATH_INFORMATION structure describing the file being created.

*ThreadId* [in]
The identifier of the thread creating the file.

*GrantedAccess* [in]
A bitmask representing the File Access Rights that the thread creating the file has been granted. File Access Rights are represented by standard Windows-defined constants as follows:

| | |
|---|---|
| **FILE_READ_DATA**<br>0x001 | The right to read the file data. |
| **FILE_WRITE_DATA**<br>0x002 | The right to write data to the file. |
| **FILE_APPEND_DATA**<br>0x004 | The right to append data to the file. |
| **FILE_READ_EA**<br>0x008 | The right to read extended file attributes. |

| | |
|---|---|
| **FILE_WRITE_EA**<br>0x010 | The right to write extended file attributes. |
| **FILE_EXECUTE**<br>0x020 | For a native code file, the right to execute the file. This access right given to scripts may cause the script to be executable, depending on the script interpreter. |
| **FILE_READ_ATTRIBUTES**<br>0x080 | The right to read file attributes. |
| **FILE_WRITE_ATTRIBUTES**<br>0x100 | The right to write file attributes. |

*CreateAction* [in]
A value indicating the action taken as a result of the thread's **CreateFile** call. The *CreateAction* will be one of the following constant values:

| | |
|---|---|
| **POL_CREATE_ACTION_SUPERSEDED**<br>0x000 | An existing file was deleted and a new file was created in its place. |
| **POL_CREATE_ACTION_OPENED**<br>0x001 | An existing file was opened. |
| **POL_CREATE_ACTION_CREATED**<br>0x002 | A new file was created. |
| **POL_CREATE_ACTION_OVERWRITTEN**<br>0x003 | An existing file was overwritten. |

See the Remarks section for additional information regarding the meaning of these parameters.

## Return value

The *PolGetPolicyNewFile* callback function returns an enumeration value of the type **FE_POLICY_RESULT** indicating the encryption policy for a new file that is being created.

To cause the file to be created with encrypted data, *PolGetPolicyNewFile* returns the enumeration value **FE_POLCY_ENCRYPT_DECRYPT**. To cause the file to be created with non-encrypted (raw, cleartext) data, *PolGetPolicyNewFile* returns the enumeration value **FE_POLICY_RAW**.

If *PolGetPolicyNewFile* cannot specify an encryption policy for the newly created file, for example due to an error in processing, *PolGetPolicyNewFile* returns the enumeration value **FE_POLICY_FAIL**. This will cause the thread's **CreateFile** operation to fail with an error.

## Remarks

All Policy DLLs must implement this callback function.

A Policy DLL's *PolGetPolicyNewFile* callback function is called by FESF to determine the policy for a new file. Policy defines whether the data written by the thread indicated by *ThreadID* will be stored encrypted or unencrypted. The policy for a given file can be based on the parameters passed into this function as well as any additional information *PolGetPolicyNewFile* acquires on its own.

Given the *ThreadID* provided in this callback, a Solution can call FESF-provided helper functions to retrieve information about the calling thread, including the directory and file name of the executing program and the Security Identifier (SID) of the account under which the thread is running. Given the SID, the Solution can get the associated username (for example, one way to do this in C++ is by calling the Windows functions *ConvertStringSidToSid* and *LookupAccountSid*).

Whenever *PolGetPolicyNewFile* returns **FE_POLCY_ENCRYPT_DECRYPT**, FESF will call the Policy DLL at its *PolGetKeyNewFile* callback function to retrieve the Algorithm ID, Key, and Policy DLL Header Data for the file.

It is important to understand that FESF considers all of the following "new files" and will therefore call *PolGetPolicyNewFile* when opening:

- A file on a supported file system that previously did not exist. In this case CreateAction will be **POL_CREATE_ACTION_CREATED**.
- A file on a supported file system that is not encrypted by FESF and is zero data bytes in length. . In this case CreateAction will be **POL_CREATE_ACTION_OPENED.**
- A file on a supported file system that is either encrypted or not encrypted by FESF and is the subject of a "destructive create." A destructive create is one with a *CreateAction* of **POL_CREATE_ACTION_SUPERSEDED** or **POL_CREATE_ACTION_OVERWRITTEN**.

The POL_CREATE_ACTION_* values are direct translations of Windows native FILE_* *CreateDisposition* values. You can read more about the specific meaning of each of these values in the MSDN documentation for the Windows **NtCreateFile** function. This documentation makes clear the difference between, for example, **POL_CREATE_ACTION_SUPERSEDED** and **POL_CREATE_ACTION_OVERWRITTEN**.

When FESF Policy Caching is enabled, if the process owning the *ThreadID* opens this file in the future with the same access described by *GrantedAccess*, FESF may automatically grant **FE_POLCY_ENCRYPT_DECRYPT** access to this file. This can help to reduce system overhead, by avoiding a future call to *PolGetPolicyExistingFile*. For more information, see FESF Policy Caching.

*PolGetPolicyNewFile* is called as part of Windows' processing a **CreateFile** call made by the thread indicated by *ThreadID* for the file described by *PolicyPathInfo*. The call to *PolGetPolicyNewFile* occurs after **CreateFile** has succeeded but before the final result is returned to the thread. Because the *PolGetPolicyNewFile* callback is blocking **CreateFile** from completing, processing in this function must be prompt.

Policy DLLs should return **FE_POLICY_FAIL** from their *PolGetPolicyNewFile* callback only when absolutely necessary. Because *PolGetPolicyNewFile* is called after Windows **CreateFile** processing has completed, returning **FE_POLICY_FAIL** will result in a new zero length file being created or, perhaps, a previously existing file being overwritten with a new zero length file. The specific action taken as a result of CreateFile is indicated by *CreateAction*. For more information see **Returning Failures from Policy Callbacks**.

## Examples

## See also

The FESF Sample Solution contains an example implementation of this callback function. This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolGetKeyNewFile callback function

A Policy DLL's *PolGetKeyNewFile* callback function provides key material for a new file that is to be stored in encrypted format.

## Syntax

```
POL_GET_KEY_NEW_FILE PolGetKeyNewFile;

bool
PolGetKeyNewFile(
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_  DWORD ThreadId,
    _Out_ PVOID *PolHeaderData,
    _Out_ DWORD *PolHeaderDataSize,
    _Out_ LPCWSTR *PolUniqueAlgorithmId,
    _Out_ PVOID *PolKey,
    _Out_ DWORD *PolKeySize,
    _Outptr_result_maybenull_ PVOID *CleanupInfo,
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the file being created.

*ThreadId* [in]

The identifier of the thread creating the file.

*PolHeaderData* [out]

A pointer to a storage area allocated by the Policy DLL containing Policy DLL defined Header Data for FESF to store with the newly created file. FESF calls the Policy DLL's *PolFreeHeader* callback function when it no longer needs the header data.

*PolHeaderDataSize* [out]

The size, in bytes, of the header data returned in the buffer pointed to by *PolHeaderData*.

*PolUniqueAlgorithmId* [out]

A pointer to a wide character string representing the encryption algorithm and properties to be used to encrypt and decrypt file data. *PolUniqueAlgorithmId* must match one previously specified by the Policy DLL in the **FE_POLICY_CONFIG** structure passed to **FePolicySetConfiguration**.

*PolKey* [out]
A pointer to a storage area allocated by the Policy DLL containing an encryption key data. FESF will pass this key data to the CNG cryptographic algorithm provider indicated by *Algorithm*. FESF calls the Policy DLL's *PolFreeKey* callback function when it no longer needs the key data and the allocated storage can be freed.

*PolKeySize* [out]
The size, in bytes, of the key in the buffer pointed to by *PolKey*.

*CleanupInfo* [out, opt]
A pointer to a Policy DLL defined context value that will be passed to the Policy DLL's *PolReportLastHandleClose* callback function. This parameter is optional and may be NULL.

## Return value

To indicate success, and that it is supplying valid values for all output parameters, the *PolGetPolicyNewFile* callback function returns **true**. Otherwise, it returns **false**.

If **false** is returned, the thread's **CreateFile** operation will fail with an error. See the Remarks section for more information on the consequences of returning **false**.

## Remarks

All Policy DLLs must implement this callback function.

A Policy DLL's *PolGetKeyNewFile* callback function is called by FESF to determine the encryption algorithm and key data for a new file that will be stored in encrypted format. This function is always called after a call to *PolGetPolicyNewFile* has returned **FE_POLICY_ENCRYPT_DECRYPT**.

Prior to the first write to a file stored in FESF encrypted format, FESF stores the Policy DLL Header Data returned in *PolHeaderData* from this function as well as certain FESF control information in the file. FESF stores the Header Data exactly as it is provided by the Policy DLL. FESF does not encrypt this data.

Once a file's data has been encrypted by FESF, when the file is opened for encrypt/decrypt access and FESF does not already have key information for the file, the Header Data will be retrieved from the file and returned to the Policy DLL at its *PolGetKeyFromHeader* callback function. Given this Header Data the Policy DLL must be able to derive the same encryption Algorithm ID, Key Data, and Key Size that are returned here

in the *PolUniqueAlgorithmId*, *PolKey*, and *PolKeySize* parameters. See the description of the *PolGetKeyFromHeader* callback function for more information.

Note that encryption key data is interpreted by FESF as a transparent block of data that it passes to the CNG provider, for use as an encryption key. For custom CNGs this transparent key data could be an indirect reference to something maintained by the custom CNG that provides the actual symmetric key used to encrypt and decrypt the file's data.

The *CleanupInfo* parameter should be set to NULL and is reserved for future use.

*PolGetKeyNewFile* is called as part of Windows' processing a system service call (such as **CreateFile**) made by the thread indicated by *ThreadID* for the file described by *PolicyPathInfo*. The call to *PolGetKeyNewFile* occurs after the system service call has succeeded but before the final result is returned to the thread. Because the *PolGetKeyNewFile* callback is blocking the system service from completing, processing in this function must be prompt.

Policy DLLs should return **false** from their *PolGetKeyNewFile* callback only when absolutely necessary. Because *PolGetKeyNewFile* is called after Windows system service processing has completed, returning **false** will result in a new zero length file being created or, perhaps, a previously existing file being overwritten with a new zero length file. For more information, see **Returning Failure from Policy Callbacks**.

## Examples

## See also

The FESF Sample Solution contains an example implementation of this callback function. This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolGetPolicyExistingFile callback function

A Policy DLL's *PolGetPolicyExistingFile* callback function determines whether a specific open instance of an existing encrypted file should receive encrypted or non-encrypted (raw, cleartext) access.

## Syntax

```
POL_GET_POLICY_EXISTING_FILE PolGetPolicyExistingFile;

FE_POLICY_RESULT
PolGetPolicyExistingFile(
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_  DWORD ThreadId,
    _In_  PVOID PolHeaderData,
    _In_  DWORD PolHeaderDataSize,
    _In_  DWORD GrantedAccess,
    _In_  DWORD CreateAction
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated FE_POLICY_PATH_INFORMATION structure describing the file being opened.

*ThreadId* [in]

The identifier of the thread opening the file.

*PolHeaderData* [in]

A pointer to an FESF allocated storage area containing the Policy DLL's Header Data that FESF retrieved from the file. This data was previously provided to FESF by the Policy DLL as output from a successful call to *PolGetKeyNewFile*.

*PolHeaderDataSize* [in]

The size, in bytes, of the header data provided in the buffer pointed to by *PolHeaderData*.

*GrantedAccess* [in]

A bitmask representing the File Access Rights that the thread opening the file has been granted. File Access Rights are represented by standard Windows-defined constants. See the description of the *GrantedAccess* parameter on the *PolGetPolicyNewFile* call for a list of these constants.

*CreateAction* [in]

A value indicating the action taken as a result of the thread's **CreateFile** call. See the description of the *CreateAction* parameter on the *PolGetPolicyNewFile* call for a list of these constants.

## Return value

The Policy DLL's *PolGetPolicyExistingFile* callback function returns an enumeration value of the type **FE_POLICY_RESULT** that determines FESF's action on subsequent read or write operations via the opened file handle.

If *PolGetPolicyExistingFile* returns **FE_POLCY_ENCRYPT_DECRYPT**, data read from the file will be transparently decrypted by FESF before it is returned to the reader, and data written to the file will be transparently encrypted by FESF before it is stored in the file.

If *PolGetPolicyExistingFile* returns **FE_POLCY_RAW**, read and write operations on the file will performed on the data as provided. That is, no transparent encryption or decryption of data will be take place.

If *PolGetPolicyExistingFile* cannot specify an encryption policy for the file being opened, for example due to an error in processing, *PolGetPolicyExistingFile* returns the enumeration value **FE_POLICY_FAIL**. This will cause the thread's **CreateFile** operation to fail with an error. See the Remarks section for more information on the consequences of returning **FE_POLICY_FAIL**.

## Remarks

All Policy DLLs must implement this callback function.

FESF calls a Policy DLL's *PolGetPolicyExistingFile* callback function whenever a thread successfully opens an existing file that contains FESF encrypted data. There are two exceptions:

- When FESF Policy Caching is enabled and policy information has already been cached for the combination of file, process, and access being processed.
- When the open being processed is a Transacted Open (such as the result of a thread calling **CreateFileTransacted**). Transacted Open operations are typically result in raw access being granted. See the discussion of Transacted Open operations at the section for the *PolApproveTransactedOpen* callback.

Note that *PolGetPolicyExistingFile* is never called when a file containing non-encrypted data is opened.

FESF calls a Policy DLL's *PolGetPolicyExistingFile* callback function to determine the policy for a specific **CreateFile** request on an existing encrypted file. The policy specifies whether a given open request is granted raw or encrypt/decrypt access to the data in the file. The policy decision can be based on the parameters passed into this function as well as any additional information *PolGetPolicyExistingFile* acquires on its own.

Given the *ThreadID* provided in this callback, *PolGetPolicyExistingFile* can call FESF-provided helper functions to retrieve additional information about the calling thread, including the directory and file name of the executing program and identity and security information of the user account under which the thread is running.

If FESF Policy Caching is enabled, FESF will remember the policy decision returned by *PolGetPolicyExistingFile*. In this case, if the process owning the *ThreadID* opens this file in the future with the same access described by *GrantedAccess*, FESF will automatically apply the same policy. This helps reduce system overhead by potentially avoiding a call to *PolGetPolicyExistingFile*. The duration of this caching behavior lasts as long as the Windows file cache for this file persists, which in turn depends on numerous factors that cannot be directly controlled. The FESF Policy Cache can be flushed at any time by the Policy DLL. For more information, see FESF Policy Caching.

*PolGetPolicyExistingFile* is called as part of Windows' processing a system service call (such as **CreateFile**) made by the thread indicated by *ThreadID* for the file described by *PolicyPathInfo*. The call to *PolGetPolicyExistingFile* occurs after the system service call has succeeded but before the final result is returned to the thread. Because the *PolGetPolicyExistingFile* callback is blocking the system service call from completing, processing in this function must be prompt.

Because of the asynchronous nature of Windows, in some unusual cases the values provided for *CreateAction* can be unexpected. For example, *PolGetPolicy**New**File* is called when an existing zero length file is encountered (including a file that is superseded as part of being opened). However, if two threads open the same file at the same time, and agree to share write access to that file, it is possible for *PolGetPolicy**Exisiting**File* to be called with *CreateAction* set to **POL_CREATE_ACTION_SUPERSEDED**, and there to be no user data in the file. The file will, however, include Policy DLL Header Data and FESF control information. While it is rare for two threads to share write access to the same file, for one of those thread to supersede its contents, and for the two threads to attempt to open this shared file at almost exactly the same time, this is possible. Policy DLLs should therefore not rely on *CreateAction* always being **POL_CREATE_ACTION_OPENED** when *PolGetPolicyExistingFile* is called.

Also note that there is an inherent risk in providing mixed responses to this call. If a given file is opened for shared write access and one open is granted **FE_POLCY_ENCRYPT_DECRYPT** and the other open is granted **FE_POLCY_RAW**, there is no way for FESF to ensure that the resulting file, with its potential mixture of encrypted and decrypted data, contains usable data. In fact, this can even lead to Client Header Data or FESF metadata being corrupted. This can lead to a situation in which FESF will identify the file as being inconsistent. See the description of the *PolReportFileInconsistent* callback for more details.

Policy DLLs should return **FE_POLICY_FAIL** from their *PolGetPolicyExistingFile* callback only when absolutely necessary. Because *PolGetPolicyExistingFile* is called after Windows **CreateFile** processing has completed, returning **FE_POLICY_FAIL** can result in a previously existing file being superseded of overwritten with a new zero length file. The specific action taken as a result of CreateFile is indicated by *CreateAction*. For more information, see **Returning Failure from Policy Callbacks**.

## Examples

## See also

The FESF Sample Solution contains an example implementation of this callback function.  This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolGetPolicyDirectoryListing callback function

In the current and all previous versions of FESF, whenever a program queries the size of a specific file using a handle that has been given RAW access to the file, the program gets back the raw (that is, uncorrected) file size. This size includes the size of the file's data, plus the FESF Metadata including the Solution Header. Similarly, in the current and all previous versions of FESF, when a program queries the size of a file using a handle that's been given ENC/DEC access to the file, FESF returns the corrected size of the file. This size reflects just the size of the data in the file.

A Policy DLL's *PolGetPolicyDirectoryListing* callback function determines whether the sizes of FESF encrypted files returned in a directory listing will reflect the raw (uncorrected) or the corrected size of any FESF encrypted files that are within that directory. Notice that this callback applies specifically to size returned in the directory listing, not the size returned to an application when it opens a file and then queries the size of that file.

Prior to FESF V1.1, when a directory was queried, FESF always returned the corrected size of the file, just like a program would get if it had ENC/DEC access and queried the size of an individual file.

Whether the raw or corrected size is returned in a directory listing can be important when applications with RAW access use the size from the directory listing (as opposed to the size from an individual query for the file size using a RAW handle) to determine how much of a file to copy. During extended usage testing of FESF V1, we were surprised to find a number of applications that did this. One specific example we found is the Microsoft OneDrive application. However, we wouldn't be surprised if other programs such as certain backup applications behaved similarly.

## Syntax

```
POL_GET_POLICY_DIRECTORY_LISTING PolGetPolicyDirectoryListing;

FE_POLICY_RESULT
PolGetPolicyDirectoryListing (
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_  DWORD ThreadId
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated FE_POLICY_PATH_INFORMATION structure describing the directory being opened.

*ThreadId* [in]

The identifier of the thread opening the directory.

## Return value

The Policy DLL's *PolGetPolicyDirectoryListing* callback function returns an enumeration value of the type **FE_POLICY_RESULT** that determines FESF's action on directory enumerations via the opened file handle.

If *PolGetPolicyDirectoryListing* returns **FE_POLICY_ENCRYPT_DECRYPT**, then the file sizes returned, if this handle is used to enumerate the directory, will be those visible to an application given **FE_POLICY_ENCRYPT_DECRYPT** in response to a call to *PolGetPolicyExistingFile*.

If *PolGetPolicyDirectoryListing* returns **FE_POLICY_RAW**, then the enumeration will return the on disk size, which is the size visible to an application given **FE_POLICY_RAW** in response to a call to *PolGetPolicyExistingFile*.

If *PolGetPolicyDirectoryListing* cannot specify an encryption policy for the file being opened, for example due to an error in processing, *PolGetPolicyDirectoryListing* returns the enumeration value **FE_POLICY_FAIL**. This will cause the thread's **CreateFile** operation to fail with an error. See the Remarks section for more information on the consequences of returning **FE_POLICY_FAIL**.

## Remarks

This function does not have to be implemented by Policy DLLs. If it is not implemented, then all directory enumerations will see the encrypted size (as if **FE_POLICY_ENCRYPT_DECRYPT** was always returned) which was the behavior in versions of FESF prior to V1.1.

FESF calls a Policy DLL's *PolGetPolicyDirectoryListing* callback function whenever a thread successfully opens a directory. Its primary use is to ensure that processes get a coherent view of a file system – thus a process which always sees the **RAW** view of all files should see the **RAW** sizes of files in a directory enumeration. This is particularly important since some applications use the sizes returned in an operation to govern how much data they consume and if they saw the **ENCDEC** size then if the file was copied raw it would be truncated.

Given the *ThreadID* provided in this callback, *PolGetPolicyDirectoryListing* can call FESF-provided helper functions to retrieve additional information about the calling thread, including the directory and file name of the executing program and identity and security information of the user account under which the thread is running.

If FESF Policy Caching is enabled, FESF will remember the policy decision returned by *PolGetPolicyDirectoryListing*. In this case, if the process owning the *ThreadID* opens this file in the future FESF will automatically apply the same policy. This helps reduce system overhead by potentially avoiding a call to *PolGetPolicyDirectoryListing*. The duration of this caching behavior lasts as long as the Windows file cache for this file persists, which in turn depends on numerous factors that cannot be directly controlled. The FESF Policy Cache can be flushed at any time by the Policy DLL. For more information, see FESF Policy Caching in the Solution Developer's Guide.

*PolGetPolicyDirectoryListing* is called as part of Windows' processing a system service call (such as **CreateFile**) made by the thread indicated by *ThreadID* for the file described by *PolicyPathInfo*. The call to *PolGetPolicyExistingFile* occurs after the system service call has succeeded but before the final result is returned to the thread. Because the *PolGetPolicyExistingFile* callback is blocking the system service call from completing, processing in this function must be prompt.

Policy DLLs should return **FE_POLICY_FAIL** from their *PolGetPolicyDirectoryListing* callback only when absolutely necessary.

### Examples

### See also

### Requirements

| | |
|---|---|
| Software version | FESF V1.1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolGetKeyFromHeader callback function

A Policy DLL's *PolGetKeyFromHeader* callback function returns the key and encryption algorithm for an existing encrypted file, given the accessing thread ID, the file path and policy header data.

### Syntax

```
POL_GET_KEY_FROM_HEADER PolGetKeyFromHeader;

bool
PolGetKeyFromHeader(
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_  DWORD ThreadId,
    _In_  PVOID PolHeaderData,
    _In_  DWORD PolHeaderDataSize,
    _Out_ LPCWSTR *PolUniqueAlgorithmId,
    _Out_ PVOID *PolKey,
```

```
    _Out_ DWORD *PolKeySize,
    _Outptr_result_maybenull_ PVOID *CleanupInfo,
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the file being created.

*ThreadId* [in]

The identifier of the thread creating the file.

*PolHeaderData* [in]

A pointer to an FESF allocated storage area containing the Policy DLL Header Data that FESF retrieved from the file. This data was previously provided to FESF by the Policy DLL as output from a successful call to *PolGetKeyNewFile*.

*PolHeaderDataSize* [in]

The size, in bytes, of the header data provided in the buffer pointed to by *PolHeaderData*.

*PolUniqueAlgorithmId* [out]

A pointer to a wide character string representing the encryption algorithm and properties to be used to encrypt and decrypt file data. *PolUniqueAlgorithmId* must match one previously specified by the Policy DLL in the **FE_POLICY_CONFIG** structure passed to **FePolicySetConfiguration**.

*PolKey* [out]

A pointer to a storage area allocated by the Policy DLL containing an encryption key data. FESF will pass this key data to the CNG cryptographic algorithm provider indicated by *Algorithm*. FESF calls the Policy DLL's *PolFreeKey* callback function when it no longer needs the key data and the allocated storage can be freed.

*PolKeySize* [out]

The size, in bytes, of the key data in the buffer pointed to by *PolKey*.

*CleanupInfo* [out, opt]

A pointer to a Policy DLL defined context value that will be passed to the Policy DLL's *PolReportLastHandleClose* callback function. This parameter is optional and may be NULL.

## Return value

To indicate success, and that it is supplying valid values for all output parameters, the *PolGetKeyFromHeader* callback function returns **true**. Otherwise, it returns **false**.

If **false** is returned, the thread's **CreateFile** operation will fail with an error. See the Remarks section for more information on the consequences of returning **false**.

## Remarks

All Policy DLLs must implement this callback function.

A Policy DLL's *PolGetKeyFromHeader* callback function is called by FESF to determine the encryption algorithm and key data to be used by the CNG provider to encrypt or decrypt data in an existing FESF encrypted file. This function is always called after a call to *PolGetPolicyExistingFile* has returned **FE_POLICY_ENCRYPT_DECRYPT** and FESF does not already have key information for the file.

As previously described, encryption key data is interpreted by FESF as an opaque data block that it passes to the CNG provider, for use as an encryption key. FESF does not interpret or verify this key data. For custom CNGs this transparent key data could be an indirect reference to something maintained by the custom CNG that provides the actual symmetric key used to encrypt and decrypt the file's data.

The Policy DLL derives the encryption algorithm and key data from the provided thread, header and **FE_POLICY_PATH_INFORMATION** for the file.

The *CleanupInfo* parameter is reserved for future use.

*PolGetKeyFromHeader* is called as part of Windows' processing a **CreateFile** call made by the thread indicated by *ThreadID* for the file described by *PolicyPathInfo*. The call to *PolGetKeyFromHeader* occurs after **CreateFile** has succeeded but before the final result is returned to the thread. Because the *PolGetKeyFromHeader* callback is blocking **CreateFile** from completing, processing in this function must be prompt.

Policy DLLs should return **false** from their *PolGetKeyFromHeader* callback only when absolutely necessary. Because *PolGetKeyFromHeader* is called after Windows **CreateFile** processing has completed, returning **false** will result in a new zero length file being created or, perhaps, a previously existing file being overwritten with a new zero length file. For more information, see **Returning Failure from Policy Callbacks**.

## Examples

## See also

The FESF Sample Solution contains an example implementation of this callback function. This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (and later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolApproveRename callback function

A Policy DLL's *PolApproveRename* callback function is called to allow the Policy DLL to approve or reject a file rename operation taking place on a supported file system.

## Syntax

```
POL_APPROVE_RENAME PolApproveRename;

bool
PolApproveRename(
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_  DWORD ThreadId,
    _In_  FE_POLICY_PATH_INFORMATION *NewPolicyPathInfo
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the original name of the file being renamed.

*ThreadId* [in]

The identifier of the thread attempting to rename the file.

*NewPolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the proposed new name of the file being renamed.
Note that if Windows has not been able supply the new name then the `FE_POLICY_PATH_TARGET_NAME_INVALID` flag will be set and the `RelativePath` will be invalid.
This is often provoked by renaming a file “through” a directory symbolic link to a network Share

## Return value

To approve the rename operation and let it proceed, the *PolApproveRename* callback function returns **true**. Otherwise, it returns **false**.

If **false** is returned, FESF will fail the thread's rename operation. How the requesting thread/process reacts to having this error returned is dependent on the thread/process.

## Remarks

A Policy DLL's *PolApproveRename* callback function is called by FESF to allow the Policy DLL to approve or reject a proposed rename operation for security reasons.
Implementation of this callback function is optional for a Policy DLL. If this function is not implemented, FESF uses the behavior specified during initialization in the field **OfflineBehavior.ApproveRename**.

Solution developers should be VERY cautious about returning "false" from this callback as a result of what should be non-fatal errors. For example, if you return "false" (thereby disallowing the requested rename operation) as a result of a function such as **GetExecutablePathForThreadId** or **GetSidForThreadId** failing due to a system protection issue, important Windows system activities such as Windows Update can fail. We advise you to use caution and good engineering judgement.

## Examples

## See also

The FESF Sample Solution contains an example implementation of this callback function. This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolApproveCreateLink callback function

A Policy DLL's *PolApproveCreateLink* callback function is called to allow the Policy DLL to approve or reject the creation of a hard link on a supported file system.

## Syntax

```
POL_APPROVE_CREATE_LINK PolApproveCreateLink;

bool
PolApproveCreateLink(
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_  DWORD ThreadId,
    _In_  FE_POLICY_PATH_INFORMATION *LinkPolicyPathInfo
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the name of the file to which the hard link is being created.

*ThreadId* [in]

The identifier of the thread attempting to create the hard link.

*LinkPolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the proposed new hard link name.
Note that if Windows has not been able supply the new name then the `FE_POLICY_PATH_TARGET_NAME_INVALID` flag will be set and the `RelativePath` will be invalid.
This is often provoked by creating a link "through" a directory symbolic link to a network Share

## Return value

To approve the creation of the hard link, the *PolApproveCreateLink* callback function returns **true**. Otherwise, it returns **false**.

If **false** is returned, FESF will fail the thread's **CreateHardLink** operation. How the requesting thread/process reacts to having this error returned is dependent on the thread/process.

## Remarks

A Policy DLL's *PolApproveCreateLink* callback function is called by FESF to allow the Policy DLL to approve or reject the creation of a hard link on a supported file system.
Implementation of this callback function is optional for a Policy DLL. If this function is not implemented, FESF uses the behavior specified during initialization in the field **OfflineBehavior.ApproveCreateLink**.

Solution developers should be VERY cautious about returning "false" from this callback as a result of what should be non-fatal errors. For example, if you return "false" (thereby disallowing the requested hardlink operation) as a result of a function such as **GetExecutablePathForThreadId** or **GetSidForThreadId** failing due to a system protection issue, important Windows system activities such as Windows Update can fail. We advise you to use caution and good engineering judgement.

## Examples

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolApproveTransactedOpen callback function

A Policy DLL's *PolApproveTransactedOpen* callback function is called to allow the Policy DLL to approve or reject a transacted open. All such opens are given RAW access and this call provides a mechanism to "veto" such raw access.

## Syntax

```
POL_APPROVE_TRANSACTED_OPEN PolApproveTransactedOpen;

bool
PolApproveTransactedOpen (
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
     _In_ LPGUID TransactionUnitOfWork,
    _In_  DWORD ThreadId,
    _In_ DWORD GrantedAccess,
    _In_ DWORD CreateAction
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the name of the file to be transactionally opened (for example, by a user having called the Windows function **CreateFileTransacted**).

*TrannsactionUnitOfWork* [in]

The ID (unit of work) of the transaction which this create is part of

*ThreadId* [in]

The identifier of the thread attempting to open the file.

*GrantedAccess* [in]

A bitmask representing the File Access Rights that the thread opening/creating the file/directory has been granted. See *PolGetPolicyExistingFile* for more details

*CreateAction* [in]

A value indicating the action taken as a result of the thread's **CreateFile** call. See the description of the *CreateAction* parameter on the *PolGetPolicyNewFile* call for a list of these constants.

## Return value

To approve the transactional open, the *PolApproveTransactedOpen* callback function returns **true**. Otherwise, it returns **false**.

If **false** is returned, FESF will fail the open. How the requesting thread/process reacts to having this error returned is dependent on the thread/process.

## Remarks

A Policy DLL's *PolApproveTransactedOpen* callback function is called by FESF to allow the Policy DLL to approve or reject the transactional open of a file.

Implementation of this callback function is optional for a Policy DLL. If this function is not implemented, FESF will allow transactions, which is the FESF behavior prior to FESF V1.1.

Solution developers should be VERY cautious about returning "false" from this callback. Many critical parts of the system (notably Windows Update) rely on transacted opens and failing them will (at best) cause such operations to fail.

Note that if a file was created as a result of a transacted open, then failing the transaction will not cause the file to be deleted. It is to be expected, however, that the application issuing the transacted open will roll back the transaction and that will have the effect of deleting the file (which was only visible within the transaction anyway).

## Examples

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1.1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolReportFileInconsistent callback function

A Policy DLL's *PolReportFileInconsistent* callback function is called to inform the Policy DLL FESF has encountered an encrypted file that is in incorrect or invalid format.

## Syntax

```
POL_REPORT_FILE_INCONSISTENT PolReportFileInconsistent;

bool
PolReportFileInconsistent(
    _In_  FE_POLICY_PATH_INFORMATION *PolicyPathInfo
)
{ ... }
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE_POLICY_PATH_INFORMATION** structure describing the name of the file that FESF has found to contain errors.

## Return value

To grant the caller raw access to the corrupt file, the *PolReportFileInconsistent* callback function returns **true**. Otherwise, it returns **false**.

If **false** is returned, FESF will fail the thread's **CreateFile** operation. How the requesting thread/process reacts to having this error returned is dependent on the thread/process.

## Remarks

A Policy DLL's *PolReportFileInconsistent* callback function is called by FESF to inform the Policy DLL that an FESF encrypted file has been found that has in invalid or inconsistent format. The Policy DLL may choose to ask FESF to attempt to recover this file, by calling the appropriate function supplied by the FESF Data Storage Service function.

Files may be reported as being inconsistent for a number of reasons. Generally, a file is considered inconsistent when FESF is reasonably sure that the file is encrypted but the file has failed one or more internal sanity checks. For example, this can happen if an application modifies certain portions of a file after being granted FE_POLICY_RAW access. It can also occur if the file is modified outside of FESF, such as when FESF is not installed on a system when the file is accessed for write. Inconsistent files are typically not safely usable until they have been repaired.

Implementation of this callback function is optional for a Policy DLL.  If this function is not implemented, FESF uses the behavior specified during initialization in the field **OfflineBehavior.ApproveCorruptFileAccess**.

## Examples

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolReportLastHandleClosed callback function

A Policy DLL's *PolReportLastHandleClosed* callback function is called to inform the Policy DLL that the last handle to a given file has been closed.

## Syntax

```
POL_REPORT_LAST_HANDLE_CLOSED PolReportLastHandleClosed;

VOID
PolReportLastHandleClosed(
    _In_  PVOID CleanupInfo
)
{ ... }
```

## Parameters

*CleanupInfo* [in]

A Policy DLL defined value that was supplied to either *PolGetKeyNewFile* or *PolGetKeyFromHeader*. This value can be used to identify the file being closed.

## Return value

(none)

## Remarks

A Policy DLL's *PolReportLastHandleClosed* callback function is called by FESF to inform the Policy DLL that the last handle is being closed on a file. This callback allows the Policy DLL to trigger additional processing of the file. Note that in FESF V1.3 and earlier, when the *PolReportLastHandleClosed* callback function is called, the indicated file is still open. Therefore, if any processing is done in *PolReportLastHandleClosed*, that processing is typically restricted to queuing work that will be done later by the Solution.

Implementation of this callback function is optional for a Policy DLL, and it is rarely specified. If this function is not implemented, FESF does not notify the Policy DLL of the last close that takes place for a given file.

## Compatibility Note

Starting in FESF V1.4, we anticipate that *PolReportLastHandleClosed* will be called after the last handle has been completely closed. This change will make it more likely, but by no means guaranteed, that a Solution that attempts to open the file in the context of this callback will succeed without encountering a sharing violation.

Note that, even with this change, there will be no guarantee that another handle hasn't been opened prior to (or during) the call to *PolReportLastHandleClosed*. Hence your Solution needs to be constructed to gracefully handle this possibility.

## Examples

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolFreeHeader callback function

A Policy DLL's *PolFreeHeader* callback function is called to enable the Policy DLL to return the storage that it previously allocated for Policy DLL Header Data.

## Syntax

```
POL_FREE_HEADER PolFreeHeader;

VOID
PolFreeHeader(
    _In_  PVOID PolHeaderData,
    _In_  DWORD PolHeaderDataSize
)
{ ... }
```

## Parameters

*PolHeaderData* [in]
A pointer to a Header Data area to be returned that was previously allocated by the Policy DLL.

*PolHeaderDataSize* [in]
The size, in bytes, of the Header Data area.

## Return value

(none)

## Remarks

A Policy DLL's *PolFreeHeader* callback function is called by FESF to allow the Policy DLL to deallocate space that it previously allocated for storage of Policy DLL Header Data. This Header Data was provided to FESF by the Policy DLL on return from the *PolGetKeyNewFile* callback function.

Policy DLLs must implement this callback function.

## Examples

## See also

The FESF Sample Solution contains an example implementation of this callback function.  This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolFreeKey callback function

A Policy DLL's *PolFreeKey* callback function is called to enable the Policy DLL to return the storage that it previously allocated for key storage.

## Syntax

```
POL_FREE_KEY PolFreeKey;

VOID
PolFreeKey(
    _In_  PVOID PolKey,
    _In_  DWORD PolKeySize
)
{ ... }
```

## Parameters

*PolKey* [in]
: A pointer to a key data storage area to be returned that was previously allocated by the Policy DLL.

*PolKeySize* [in]
: The size, in bytes, of the key storage area.

## Return value

(none)

## Remarks

A Policy DLL's *PolFreeKey* callback function is called by FESF to allow the Policy DLL to deallocate space that it previously allocated for storage of key data information. This key buffer was provided to FESF by the Policy DLL on return from the *PolGetKeyNewFile* or *PolGetKeyFromHeader* callback function.

This callback function is separate from the *PolFreeHeader* callback function to allow for different allocation and return methods for Header Data (which is presumably not security sensitive) and key information (which is presumably sensitive from a security standpoint). In most Policy DLL implementations *PolFreeKey* would overwrite the key storage area with a random data before freeing it.

Policy DLLs must implement this callback function.

## Examples

## See also

The FESF Sample Solution contains an example implementation of this callback function.  This example is part of the provided UM_Sample Visual Studio Solution, the SampPolicy project, and is located in the file SampPolicy.cpp.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# PolUnInit callback function

A Policy DLL's *PolUnInit* callback function is called when the system shuts down.

## Syntax

```
POL_UNINIT PolUnInit;

VOID
PolUnInit(
    VOID
)
{ ... }
```

## Parameters

(none)

## Return value

(none)

## Remarks

A Policy DLL's *PolUnInit* callback function is called by FESF to allow the Policy DLL to perform an orderly tear down of any state.

Policy DLLs need not implement this callback function.

## Examples

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# FESF Policy Function Reference

The functions in this section are implemented by the FESF Policy Service for exclusive use of the Client's Solution Policy DLL.

# FePolSetConfiguration function

Called by the Policy DLL to provide its desired configuration parameters to FESF.

## Syntax

```
DWORD
FePolSetConfiguration(
    _In_  FE_POLICY_CONFIG * Configuration
)
{ ... }
```

## Parameters

*Configuration* [in]

A pointer to a **FE_POLICY_CONFIG** structure that has been filled-in by the Policy DLL to reflect its desired configuration.

## Return value

If the function succeeds, **ERROR_SUCCESS** is returned.

If the function fails for any reason, an appropriate error code is returned. If a pointer to a required function is missing from the **FE_POLICY_CONFIG** structure, **ERROR_INVALID_DATA** is returned.

## Remarks

Every Policy DLL must call this function from within its **PolicyDllInit** callback function, to establish the desired configuration and provide pointers to other Policy DLL functions for the FESF Policy Service to call.

Note that the Solution Policy DLL can start to receive callbacks from FESF as soon as this call is made.

## Examples

The following example illustrates setting up the FE_POLICY_CONFIG structure and calling **FePolicySetConfiguration** within the Policy DLL's *PolicyDllInit* callback function.

```
PolicyDllInit(VOID)
{
    FE_POLICY_CONFIG              config = {0};
    FE_POLICY_ALGORITHM           algorithm;
    FE_POLICY_ALGORITHM_PROPERTY *aesProperty;
    DWORD                         error;
    bool                          result;

    //
    // We only use ONE algorithm, thus the size of our structure is
    // constant. This would need to be updated if the number of
    // algorithms used was > 1
    //
    config.Length       = sizeof(FE_POLICY_CONFIG);
    config.VersionMajor = FE_POLICY_VERSION_MAJOR;
    config.VersionMinor = FE_POLICY_VERSION_MINOR;

    config.PolGetPolicyNewFile       = ExamplePolGetPolicyNewFile;
    config.PolGetKeyNewFile          = ExamplePolGetKeyNewFile;
    config.PolGetPolicyExistingFile  = ExamplePolGetPolicyExistingFile;
    config.PolGetKeyFromHeader       = ExamplePolGetKeyFromHeader;
    config.PolFreeHeader             = ExamplePolFreeHeader;
    config.PolFreeKey                = ExamplePolFreeKey;

    config.OfflineBehavior.PolApproveCreateLink     = true;
    config.OfflineBehavior.PolApproveRename         = true;
    config.OfflineBehavior.ApproveCorruptFileAccess = true;
    config.OfflineBehavior.RawDirSize = false;

    config.AccessCache.Enable = true;

    config.AlgorithmsCount = 1;
    config.Algorithms[0]   = &algorithm;

    algorithm.PolUniqueAlgorithmId       = ExampleUniqueAlgorithmId;
    algorithm.CNGAlgorithmIdentifier     = BCRYPT_AES_ALGORITHM;
    algorithm.CNGAlgorithmImplementation = NULL;

    algorithm.PropertiesCount = 1;

    aesProperty = &algorithm.Properties[0];

    aesProperty->CNGPropertyIdentifier  = BCRYPT_CHAINING_MODE;
    aesProperty->CNGPropertyValue       = BCRYPT_CHAIN_MODE_CBC;
    aesProperty->CNGPropertyValueLength =
                (sizeof(BCRYPT_CHAIN_MODE_CBC) - sizeof(WCHAR));

    error = FePolSetConfiguration(&config);

    if (error == ERROR_SUCCESS) {
        result = true;
    } else {
        result = false;
    }

    return result;

}
```

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Library | FESFPolicy.lib |
| Header | PolDllApi.h |

# FESFUtility Function Reference

The functions in this section are implemented by the FESFUtility DLL.

# Using the FesfUtility DLL

The FESFUtility DLL is a COM in-process server. It exports the functions describe in this section for use by Client Solutions when FESF is installed and running on the system. Note FesfUtility exports two Interfaces as described below.

**Type Library**: \UM_FESF\UMLIB\FESFUTILITY.TLB

**CLSID:** FesfUtil {a5cf6c1a-fba3-46e9-9a06-99e3879337a3}
**IID:** IFesfUtil {287ae2ac-d46b-478a-b843-fb49d0818958}
**IID:** IFesfUtil2 {2a5ed4b-2b5d-4cca-abe4-2246c994edc3}

**Note**: Please use the definitions provided in the type library as the GUIDs are subject to change.

Starting with FESF V1.3 the Interface IFesfUtil2 is present. This Interface provides support for all the original functions available via the IFesfUtil Interface, as well the **ReadHeaderUnsafe** and **UpdateHeaderUpdate** families of functions, which are exclusively available via the IFesfUtil2 Interface.

Functions in the FesfUtility DLL can be invoked using standard COM mechanisms. This is illustrated in the following example.

In the Header file:

```
#import "..\..\UM_FESF\UMLib\FESFUTILITY.tlb" no_namespace raw_interfaces_only
IFesfUtil *m_spUtil;
```

In the executable function:

```
// Initialize COM
::CoInitializeEx(nullptr, COINIT_MULTITHREADED);

hr = ::CoCreateInstance(__uuidof(FesfUtil),
                        nullptr,
                        CLSCTX_ALL,
                        IID_PPV_ARGS(&m_spUtil));
if (FAILED(hr)) {
        // Could not find the FesfUtil DLL
        ATLTRACE(L"Failed to access FesfUtil: 0x%X\n", hr);
        return E_UNEXPECTED;
}

//
// Is the FESF Policy Service running?
//
VARIANT_BOOL running = VARIANT_FALSE;

hr = m_spUtil->IsFESFServiceRunning(&running);
```

(code example continues on next page)

```
if (!hr != S_OK)
{

    // the call FAILED?  That's odd…
    ::MessageBox(nullptr, L"Call to IsFESFServiceRunning failed?",
                          L"Testing", MB_ICONEXCLAMATION | MB_OK);
    return E_FAIL;
}

if (!running)
{
    // service is unavailable
    ::MessageBox(nullptr, L"Service is unavailable or not running",
                          L"Testing", MB_ICONEXCLAMATION | MB_OK);
}

 return S_OK;
```

# PurgePolicyCacheFile function

Purges data stored in the FESF Policy Cache for a specific file.

## Syntax

```
HRESULT
PurgePolicyCacheFile(
    [in]  BSTR FullPath
)
{ ... }
```

## Parameters

*FullPath* [in]
A string representing the fully qualified path name of the file to be purged.

## Return value

If the function succeeds, **S_OK** is returned.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

A Policy DLL may call this function to cause FESF to remove all references to a given file from its Policy Cache. Note that there may be some delay between the time this function is called and the Policy Cache is completely purged for the given file.

For more information on FESF Policy Caching see the section FESF Policy Caching in this document.

This function relies on support from the FESF Kernel Mode Components. Note that functions in the FESF Utility library are only designed for use when FESF is installed and the FESF Kernel Mode Components are running.

## Examples

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |

| | |
|---|---|
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil  (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# PurgePolicyCacheThread function

Purges data stored in the FESF Policy Cache for a specific process, given a Thread ID associated with that process.

## Syntax

```
HRESULT
PurgePolicyCacheThread(
    [in]  int ThreadId
)
{ ... }
```

## Parameters

*ThreadId* [in]

The identifier of a thread whose process is to be removed from the FESF Policy Cache, or zero to remove the cached data for all processes from the FESF Policy Cache.

## Return value

If the function succeeds, **S_OK** is returned.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

A Policy DLL may call this function to cause FESF to remove all references to a given process, or all processes, from its Policy Cache. Note that there may be some delay between the time this function is called and the Policy Cache is completely purged.

Even though this function takes a Thread ID in the *ThreadId* parameter, the FESF Policy Cache is purged for all threads in the process that owns the specified thread. If the *ThreadId* parameter is passed as zero, the FESF Policy Cache is purged for all threads and all processes.

This function relies on support from the FESF Kernel Mode Components. Note that functions in the FESF Utility library are only designed for use when FESF is installed and the FESF Kernel Mode Components are running.

For more information on FESF Policy Caching see the section FESF Policy Caching in this document.

## Examples

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil  (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# GetExecutablePathForThreadId function

Returns the executable path (including name of the image) associated with a given thread.

## Syntax

```
HRESULT
GetExecutablePathForThreadId(
    [in]           int ThreadId,
    [out, retval]  BSTR* PathBuffer
)
{ ... }
```

## Parameters

*ThreadId* [in]

The identifier of the thread for which the executable image path is being sought.

*PathBuffer* [out, retval]

A pointer to a BSTR into which to store the result on success. On return with success, a string holding the path to the executable image name associated with the thread. On return with an error status, the contents of *PathBuffer* are undefined.

## Return value

A standard **HRESULT** value indicating the success or failure of the lookup operation.

Note that this function can fail due to security reasons, when called for certain Windows protected processes. This is by (Microsoft's) design and cannot be bypassed.

## Remarks

On success, this function returns a fully qualified executable path for the executable image associated with the thread identified by the provided *ThreadId* (TID). This function relies on support from the FESF Kernel Mode Components to ensure that all TIDs, even those of protected processes, can be translated to a path.

This function will return one of four different path formats:

1. Local volume, mount point found
   e:\FileTest.exe
2. Local volume, no mount point
   \\?\Volume{xxx}\FileTest.exe
3. Network share

\\Server\Share\FileTest.exe

4. Shadow volume
   \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy9\FileTest.ext

If the thread is owned by the system process, then this function will return “System” as the path.

Note that functions in the FESF Utility library are only designed for use when FESF is installed and the FESF Kernel Mode Components are running.

## Examples

See the example shown at **GetSidForThreadId**.

## See also

The provided Sample Policy DLL (in the UM_Sample solution) contains multiple examples that illustrate the use of **GetExecutablePathForThreadId**.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# IsFileEncrypted function

Determines if a given file is stored in FESF encrypted format.

## Syntax

```
HRESULT
IsFileEncrypted(
    [in]           BSTR  FullPath,
    [out, retval]  VARIANT_BOOL* Encrypted
)
{ ... }
```

## Parameters

*FullPath* [in]

A string containing the fully qualified path of a file to check.

*Encrypted* [out, retval]

A pointer to a VARIANT_BOOL that will receive the result on success. Set to VARIANT_TRUE if the file indicated by *FullPath* is in FESF encrypted format.

## Return value

**S_OK** on success. **E_POINTER** if the *FullPath* parameter is not provided. Other standard HRESULT values may be returned indicating the failure of the operation.

## Remarks

If the file indicated by *FullPath* is encrypted, the **HRESULT** from this function will be S_OK and the value returned in *Encrypted* will be VARIANT_TRUE.

This function relies on support from the FESF Kernel Mode Components. Note that functions in the FESF Utility library are only designed for use when FESF is installed and the FESF Kernel Mode Components are running.

## Example

```
// form the fully qualified file path
CString file(basePath + findData.cFileName);
VARIANT_BOOL answer;
```

```
hr = spUtil->IsFileEncrypted(CComBSTR(file), &answer);

bool fileIsEncrypted = (answer == VARIANT_TRUE);

if (fileIsEncrypted)
{
    wprintf(L"%-20ws: is encrypted!\n",
                    wcsrchr(file, L'\\') + 1);

    continue;
}
```

## See also

The provided UM_Sample solution contains several examples that illustrates the use of this function.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# GetFullyQualifiedLocalPath function

Converts a Volume GUID and Relative Path pair into a sting containing a fully qualified path on a local volume.

## Syntax

```
HRESULT
GetFullyQualifiedLocalPath(
    [in]           REFGUID Volume,
    [in]           BSTR RelativePath,
    [out, retval]  BSTR* FullPath
)
{ ... }
```

## Parameters

*Volume* [in]

GUID representing a local volume.  This GUID may not be the FESF Network Volume GUID (FE_NETWORK_GUID) or the FESF Shadow Volume GUID (FE_SHADOW_VOLUME_GUID), , and may not be NULL or empty.

*RelativePath* [in]

A path, relative to the supplied *Volume* GUID.

*FullPath* [out, retval]

A pointer to a BSTR into which to store the result on success.

## Return value

A standard **HRESULT** value indicating the success or failure of the lookup operation.

## Remarks

If the FESF Network Volume GUID (FE_NETWORK_GUID) or the FESF Shadow Volume GUID (FE_SHADOW_VOLUME_GUID) is provided for the *Volume* argument, the function returns HRESULT E_FAIL.

On return with success, *FullPath* points to a string holding the fully qualified path for the file on a local volume.  On return with an error status, the contents of *FullPath* are undefined.

## Examples

## See also

The provided UM_Sample solution contains examples that illustrate the use of **GetFullyQualifiedLocalPath**.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# IsFesfServiceRunning function

Determines if the FESF Policy Service (FesfPolicy) is in the running state.

## Syntax

```
HRESULT
IsFesfServiceRunning(
    [out, retval]  VARIANT_BOOL* ServiceRunning
)
{ ... }
```

## Parameters

*ServiceRunning* [out, retval]
A pointer into which to return a value indicating whether FesfPolicy is running.

## Return value

If the function succeeds, **S_OK** is returned.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

A Client Solution may call this function to determine if FesfPolicy is actively running. In this case, "actively running" means that the FESF Policy Service has started, has performed its required initialization, and is actively processing requests.

## Examples

An example of how to invoke this function is provided at the beginning of this section.

## See also

The provided UM_Sample solution contains several examples that illustrate the use of this function.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |

| | |
|---|---|
| Supported FESF State | FESF Online State |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil  (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# IsThreadIdInSid function

Determines if the account under which a given thread is running is a member of a given Security Group.

## Syntax

```
HRESULT
IsThreadIdInSid(
    [in] int ThreadId,
    [in] BSTR GroupSid,
    [out, retval] VARIANT_BOOL* IsInGroup
)
{ ... }
```

## Parameters

*ThreadId* [in]
The Thread ID of a thread to be checked for group membership.

*GroupSid* [in]
A Security ID (SID) in string describing a Security Group.

*IsInGroup* [out, retval]
A pointer into which to return a VARIANT_BOOL that on successful completion indicates if the Security ID under which *ThreadId* is executing is a member of *GroupSid*.

## Return value

If the function succeeds, **S_OK** is returned.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

**IsThreadIdInSid** determines if the Security Principle under which a given thread (identified by *ThreadId*) is executing is a member of the Security Group identified by *GroupSid*.

Client Solutions can use this function to determine if the user that's running an application instance (identified by a Thread ID) is a member of a given Active Directory Security Group.  That group could be a Windows built-in group (such as "Backup Operators") or a custom-defined group (such as "MI5 Agents").

This function relies on support from the FESF Kernel Mode Components. Note that functions in the FESF Utility library are only designed for use when FESF is installed and the FESF Kernel Mode Components are running.

## Examples

```
if (m_User != user) {

    VARIANT_BOOL answer;

    m_pFESFUtil->IsThreadIdInSid(ThreadId, m_User, &answer);

    if (VARIANT_FALSE == answer) {
        return S_OK;
    }
}
```

## See also

The RuleEngine project that is part of the provided UM_Sample solution contains an example that illustrates the use of this function.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# GetSidForThreadId function

Retrieves the Security Identifier under which a given thread is running.

## Syntax

```
HRESULT
GetSidForThreadId(
    [in]           int ThreadId
    [out, retval]  BSTR* SidString
)
{ ... }
```

## Parameters

*ThreadId* [in]
: The Thread ID of a currently active thread.

*SidString* [out, retval]
: A pointer to a string into which to return the string format of the Security Identifier (SID) under which the thread is executing.

## Return value

If the function succeeds, **S_OK** is returned.

Note that this function can fail due to security reasons, when called for certain Windows protected processes. This is by (Microsoft's) design and cannot be bypassed.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

A Client Solution may call this function to retrieve the Security Identifier (SID) under which a given thread is executing.

This function relies on support from the FESF Kernel Mode Components. Note that functions in the FESF Utility library are only designed for use when FESF is installed and the FESF Kernel Mode Components are running. If **GetSidForThreadId** is called when the FESF Kernel Mode Components are not running, an error value is returned.

## Example

The following example is taken from the Sample Policy DLL (SampPolicy) that is part of the UM_Sample project.

```
//
// Get the SID (Security ID) based on the calling thread. This uniquely
// identifies the user that created the thread, so it can be used in
// user-based policy decisions. The FesfUtil dll provides a helper for this.
//
CComBSTR sid;
HRESULT hr = g_pFesfUtil->GetSidForThreadId(ThreadId, &sid);
if (FAILED(hr))
{
    ATLTRACE(L">>>Failed to obtain SID for thread %d : %x\n", ThreadId, hr);
    return FE_POLICY_RAW;
}

//
// Get a path to the executable that is creating the file. FesfUtil provides
// a helper for this as well.
//
CComBSTR exePath;
hr = g_pFesfUtil->GetExecutablePathForThreadId(ThreadId, &exePath);

if (FAILED(hr))
{
    ATLTRACE(L">>>Failed to obtain executable file for thread %d : %x\n", ThreadId, hr);
    return FE_POLICY_RAW;
}

//
// We've gathered all the sample policy manager needs to make a decision,
// so ask it what we should do with the file. We could make that decision
// here if it were more convenient to do so.
//
VARIANT_BOOL encdec = VARIANT_FALSE;
hr = g_pPolicyManager->GetPolicyNewFile(
                &PolicyPathInfo->VolumeGuid, CComBSTR(PolicyPathInfo->RelativePath),
                exePath, sid, ThreadId, &encdec);
```

## See also

The provided UM_Sample solution contains several examples that illustrate the use of this function.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State. |

| | |
|---|---|
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil  (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# GetTrueFileSize function

Retrieves the actual on-disk size of an FESF encrypted file, including the size of the Client Solution Policy Header and any FESF metadata.

## Syntax

```
HRESULT
GetTrueFileSize(
    [in]           BSTR FullPath,
    [out, retval]  __int64* TrueSize
)
{ ... }
```

## Parameters

*FullPath* [in]
A string containing the fully qualified path to an FESF encrypted file.

*TrueSize* [out, retval]
A pointer to a 64-bit integer location into which the actual on-disk size of the file is stored.

## Return value

If the function succeeds, **S_OK** is returned.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

When FESF encrypts a file, additional data is stored along with that file. This data includes both the Client Solution's Policy Header, and a small amount of FESF metadata. When file size information is retrieved by standard means, FESF corrects the file size to represent only the size of the data in the file. Therefore, the file size ordinarily does not reflect the actual number of bytes a given file occupies on disk.

A Client Solution may use the **GetTrueFileSize** function to retrieve the actual number of bytes that a file occupies on disk, including the Solution's Policy Header and FESF metadata.

The calling COM client must be able to access the file for this operation to succeed.

## Examples

## See also

The UM_FESF solution contains an example that illustrates the use of **GetTrueFileSize**.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL | FESFUtility.DLL |
| Supported FESF State | FESF Online State. |
| Type Library | \UM_FESF\UMLIB\FESFUTILITY.TLB |
| IID | IFesfUtil (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# ReadHeaderUnsafe and ReadHeaderUnsafeFQP functions

Provides a mechanism to read the header of an encrypted file, even if the file is currently in use by another process.

## Syntax

```
HRESULT
ReadHeaderUnsafe(
        [in] REFGUID VolumeGuid,
        [in] BSTR Path,
        [out, retval] VARIANT *Header
)

HRESULT
ReadHeaderUnsafeFQP(
        [in] BSTR Fqp,
        [out, retval] VARIANT *Header
)
```

## Parameters

*VolumeGuid* [in]

GUID representing a local volume.  This GUID must not be the FESF Network Volume GUID (FE_NETWORK_GUID) or the FESF Shadow Volume GUID (FE_SHADOW_VOLUME_GUID), and must not be NULL or empty.

*RelativePath* [in]

A path, relative to the supplied *Volume* GUID.

*FQP* [in]

A Fully Qualified (Windows) Path to a file on a local volume. File must not be located on a network share.

*Header* [out, retval]

The current file header

## Return value

If the function succeeds, **S_OK** is returned.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

The file must be FESF Encrypted and the caller must have SeRestorePrivilege.

## Requirements

| | |
|---|---|
| Software version | FESF V1.2.4 (or later) |
| DLL | FESFUtility.DLL |
| IID | IFesfUtil2 (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# UpdateHeaderUnsafe and UpdateHeaderUnsafeFQP functions

Provides a mechanism to update the header of an encrypted file, even if the file is currently in use by another process.

## Syntax

```
HRESULT
UpdateHeaderUnsafe(
        [in] REFGUID VolumeGuid,
        [in] BSTR Path,
        [in] VARIANT *OldHeader,
        [in] VARIANT *NewHeader
)

HRESULT
UpdateHeaderUnsafeFQP(
        [in] BSTR Fqp,
        [in] VARIANT *OldHeader,
        [in] VARIANT *NewHeader
)
```

## Parameters

*VolumeGuid*[in]

GUID representing a local volume.  This GUID must not be the FESF Network Volume GUID (FE_NETWORK_GUID) or the FESF Shadow Volume GUID (FE_SHADOW_VOLUME_GUID), and must not be NULL or empty.

*RelativePath* [in]

A path, relative to the supplied *Volume* GUID.

*FQP* [in]

A Fully Qualified (Windows) Path to a file on a local volume. File must not be located on a network share.

*OldHeader* [in]

The current file header.

*NewHeader* [in]
The new file header.

## Return value

If the function succeeds, **S_OK** is returned.

If the function fails for any reason, an appropriate error code is returned.

## Remarks

This function will change the file header of a currently open file. It is an extra-ordinarily dangerous function to call since the rest of the stack is unaware that the header has changed and so such a change must not involve a change to the encryption key – otherwise the file will have been hopelessly and irretrievably corrupted.

After a successful call to this function, the updated header is passed to the Policy DLL callbacks on subsequent opens of the file.

The call will fail if:

- The new header size is not the same as the old header size.
- The provided old header does not match the current (on disk) header.
- The file is not on a local disk.
- The caller does not have **SeRestorePrivilege**.
- The file is not FESF Encrypted.

The caller of this API is responsible for protecting a file against multiple, simultaneous header updates. The results are undefined if simultaneous calls to **UpdateHeaderUnsafe**/**UpdateHeaderUnsafeFQP** are made to a file.

## Requirements

| | |
|---|---|
| Software version | FESF V1.2.4 (or later) |
| DLL | FESFUtility.DLL |
| IID | IFesfUtil2 (please use the defintion from the Type Library) |
| CLSID | FesfUtil (please use the definition from the Type Library) |

# FESFDs Function Reference

The functions in this section are implemented by the FESF Data Storage Service (FesfDs.exe).

# Using Functions in the FesfDs Service

The FesfDs service is an out-of-process COM server. It exports the functions described in this section for use by Client Solutions in FESF Online State (when FESF is installed and running on the system) or in FESF Offline State (as long as the FesfDS Service is accessible via COM).

**Type Library**: \UM_FESF\UMLIB\FESFDS.TLB

**CLSID:** FesfDs {8a8b65c6-8862-4856-95f9-9f88b2ec785d}
**IID:** IFesfDs {8ef95237-cec8-4c30-99c0-db43e61bdea8}

**Note**: Please use the definitions provided in the type library as the GUIDs are subject to change.

Functions exported by the FesfDs Service can be invoked using standard COM mechanisms. One example of using these mechanisms is illustrated in the following example.

In the Header file:

```
#import "..\..\UM_FESF\UMLib\FESFDS.tlb" no_namespace raw_interfaces_only
```

In the executable function:

```
// Initialize COM
HRESULT hr = ::CoInitialize(0);

if (FAILED(hr))
{
    return Error(hr);
}

// get data storage object
CComPtr<IFesfDs> spDS;
HRESULT hr = spDS.CoCreateInstance(__uuidof(FesfDs));

if (FAILED(hr))
{
    return Error(hr);
}

// check if file is encrypted

VARIANT_BOOL encrypted;
hr = spDS->IsFileEncrypted(const_cast<GUID *>(&GUID_NULL),
                           CComBSTR(file),
                           &encrypted);

if (FAILED(hr))
{
    wprintf(L"Failed to determine status\n");
    return;
}

if (encrypted == VARIANT_TRUE) {

    wprintf(L"File is encrypted\n");
} else {
```

```
            wprintf(L"File is not encrypted\n");
        }
```

# IsFileEncrypted function

Determines if a given file is stored in FESF encrypted format.

## Syntax

```
HRESULT
IsFileEncrypted(
    [in]           REFGUID Volume,
    [in]           BSTR    Path,
    [out, retval]  VARIANT_BOOL* Encrypted
)
{ ... }
```

## Parameters

*Volume* [in]

A reference to a GUID that identifies the volume on which the file resides. See Remarks for further information.

*Path* [in]

A string containing the path of a file to check. This may be a fully qualified path or a path relative to the *Volume* argument. Refer to the Remarks section.

*Encrypted* [out, retval]

A pointer to a VARIANT_BOOL that will receive the result on success. Set to VARIANT_TRUE if the file indicated by *Path* is in FESF encrypted format.

## Return value

**S_OK** on success

**E_FAIL** if the file cannot be opened for read access, the required privileges are not available to the caller, or for various other fatal error conditions.

Other standard HRESULT values may be returned indicating the failure of the operation.

## Remarks

If *Volume* is equal to FE_NETWORK_GUID (indicating that *Path* refers to a file on a network volume), then *Path* is interpreted as a fully qualified path, suitable for direct evaluation.

If *Volume* is equal to FE_SHADOW_VOLUME_GUID (indicating that *Path* refers to a file on a network volume), then path is interpreted as *either* as a fully qualified path, suitable for direct evaluation *or* as the concatenation of the shadow device name and path.

If *Volume* is equal to GUID_NULL (indicating that no GUID is provided) then the path is inspected and treated as though either FE_SHADOW_VOLUME_GUID or FE_NETWORK_GUID was provided.

Otherwise, *Volume* represents a local volume and *Path* is interpreted as relative to that volume.

In most cases, the calling COM client must have read access to the file.

## Example

## See also

The provided UM_Sample solution contains examples that illustrates the use of this function.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL/Server | FesfDs.exe |
| Supported FESF State | FESF Online State or FESF Offline State (as long as FesfDs in accessible via COM). |
| IID | IFesfDs  (please use the defintion from the Type Library) |
| CLSID | FesfDs (please use the definition from the Type Library) |

# GetTrueSize function

Retrieves the total on-disk size of a file stored in FESF encrypted format.

## Syntax

```
HRESULT
GetTrueSize(
    [in]           REFGUID Volume,
    [in]           BSTR    Path,
    [out, retval]  __int64* TrueSize
)
{ ... }
```

## Parameters

*Volume* [in]

A reference to a GUID that identifies the volume on which the file resides. See Remarks for further information.

*Path* [in]

A string containing the path of a file to check. This may be a fully qualified path or a path relative to the *Volume* argument. Refer to the Remarks section.

*TrueSize* [out, retval]

A pointer to a 64-bit location that will receive the size, in bytes, of the file on success.

## Return value

**S_OK** on success

Other standard HRESULT values may be returned indicating the failure of the operation.

## Remarks

If *Volume* is equal to FE_SHADOW_VOLUME_GUID (indicating that *Path* refers to a file on a network volume), then path is interpreted as *either* as a fully qualified path, suitable for direct evaluation *or* as the concatenation of the shadow device name and path.

If *Volume* is equal to GUID_NULL (indicating that no GUID is provided) then the path is inspected and treated as though either FE_SHADOW_VOLUME_GUID or FE_NETWORK_GUID was provided.

Otherwise, *Volume* represents a local volume and *Path* is interpreted as relative to that volume.

When FESF encrypts a file, additional data is stored along with that file. This data includes both the Client Solution's Policy Header, and a small amount of FESF metadata. When file size information is retrieved by standard means, FESF corrects the file size to represent only the size of the data in the file. Therefore, the file size ordinarily does not reflect the actual number of bytes a given file occupies on disk.

A Client Solution may use the **GetTrueSize** function to retrieve the actual number of bytes that a file occupies on disk, including the Solution's Policy Header and FESF metadata.

The calling COM client must have read access to the file to call this function.

## Example

## See also

The provided UM_Sample solution contains examples that illustrates the use of this function.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL/Server | FesfDs.exe |
| Supported FESF State | FESF Online State or FESF Offline State (as long as FesfDs in accessible via COM). |
| Type Library | \UM_FESF\UMLIB\FESFDS.TLB |
| IID | IFesfDs (please use the defintion from the Type Library) |
| CLSID | FesfDs (please use the definition from the Type Library) |

# Encrypt function

This function is reserved to OSR.

## Syntax

```
HRESULT
Encrypt(
    [in] REFGUID Volume,
    [in] BSTR    Path,
    [in] BSTR    User
   )
{ ... }
```

# Decrypt function

This function is reserved to OSR.

## Syntax

```
HRESULT
Decrypt(
    [in] REFGUID Volume,
    [in] BSTR    Path,
    [in] BSTR    User
   )
{ ... }
```

# CheckFile function

Checks an FESF encrypted file to determine if it is consistent and valid, and optionally repairs that file returning it to a previously known correct state.

## Syntax

```
HRESULT
CheckFile(
    [in] REFGUID Volume,
    [in] BSTR    Path,
    [in] BSTR    User,
    [in] VARIANT_BOOL RepairIfDamaged
   )
{ ... }
```

## Parameters

*Volume* [in]

A reference to a GUID that identifies the volume on which the file resides. See Remarks for further information.

*Path* [in]

A string containing the path of a file to check. This may be a fully qualified path or a path relative to the *Volume* argument. Refer to the Remarks section.

*User* [in]

A string value identifying the user requesting the operation.

*RepairIfDamaged* [in]

If VARIANT_TRUE, FesfDs will attempt to repair the file if it is found to be inconsistent. If VARIANT_FALSE FesfDs will only report the status of the file to the caller.

## Return value

Returns **E_FAIL** or **E_NOTIMPL**.

## Remarks

This function is not implemented in FESF V1.

## Example

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| DLL/Server | FesfDs.exe |
| Supported FESF State | FESF Online State or FESF Offline State (as long as FesfDs in accessible via COM). |
| Type Library | \UM_FESF\UMLIB\FESFDS.TLB |
| IID | IFesfDs  (please use the defintion from the Type Library) |
| CLSID | FesfDs (please use the definition from the Type Library) |

# ReadHeader function

Retrieves the header of an FESF Encrypted file.

## Syntax

```
HRESULT
ReadHeader(
    [in]  REFGUID Volume,
    [in]  BSTR    Path,
    [out] ULONG *MaxHeaderLength,
    [out, retval] VARIANT *Header
   )
{ ... }
```

## Parameters

*Volume* [in]

A reference to a GUID that identifies the volume on which the file resides. See Remarks for further information.

*Path* [in]

A string containing the path of a file whose header is to be read. This may be a fully qualified path or a path relative to the *Volume* argument. Refer to the Remarks section.

MaxHeaderLength [out]

The maximum total length, in bytes, of the Header Data that can be written without having to extend the file. See Remarks for more information.

*Header* [out]

A pointer to a VARIANT into which to return, on success, the Client Solution's Policy Header Data. See the Remarks for more information.

## Return value

**S_OK** on success

Other standard HRESULT values may be returned indicating the failure of the operation.

## Remarks

A Client Solution component calls **ReadHeader** to retrieve its Policy Header Data from an FESF encrypted file. In case the Solution component wants to subsequently update the Policy Header Data, this method also returns the maximum size of a new Header Data area that may be written without having to extend the file.

If *Volume* is equal to FE_SHADOW_VOLUME_GUID (indicating that *Path* refers to a file on a network volume), then path is interpreted as *either* as a fully qualified path, suitable for direct evaluation *or* as the concatenation of the shadow device name and path.

If *Volume* is equal to GUID_NULL (indicating that no GUID is provided) then the path is inspected and treated as though either FE_SHADOW_VOLUME_GUID or FE_NETWORK_GUID was provided.

Otherwise, Volume represents a local volume and Path is interpreted as relative to that volume.

The Policy DLL's Header Data is returned to the caller in a *CComSafeArray* of bytes, described by a COM (automation) *VARIANT* structure. The header data in the returned *CComSafeArray* can be used directly or copied to a buffer, as shown in the example below.

The calling COM client must have SE_RESTORE_NAME privilege available to call this function.

## Example

```
BYTE *existingHeader{ nullptr };
BYTE *newHeader{ nullptr };
auto headerLength = (ULONG)0;
ULONG newLength;
ULONG maxHeaderLength{ 0 };
VARIANT variantHeader;
CString errorString(L"");

//
// Read the existing Header
//
hr = spDS->ReadHeader(const_cast<GUID *>(&GUID_NULL),
                      GetUniversalFilePath(file),
                      &maxHeaderLength,
                      &variantHeader);

if (!SUCCEEDED(hr))
{
    wprintf(L"%-30ws <E> ReadHeader failed 0x%x\n", findData.cFileName, hr);
    break;
}


//
// We need to convert the variant version of the header into a
// byte array version of the header, which we do via a
// safearray of bytes.
//
```

```
CComSafeArray<BYTE> headerAsSafeArray;

if (nullptr == variantHeader.parray)
{
    wprintf(L"%-30ws <E> ReadHeader returned null header (hr=0x%x)\n",
            findData.cFileName, hr);
    break;
}

headerAsSafeArray.Attach(variantHeader.parray);
headerLength = headerAsSafeArray.GetCount();
existingHeader = new BYTE[headerLength];

//
// Move the data from the safe array back into the header buffer
//
for (ULONG index = 0; index < headerLength; index++)
{
    existingHeader[index] = headerAsSafeArray.GetAt(index);
}
```

## See also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (and later) |
| DLL/Server | FesfDs.exe |
| Supported FESF State | FESF Online State or FESF Offline State (as long as FesfDs in accessible via COM). |
| Type Library | \UM_FESF\UMLIB\FESFDS.TLB |
| IID | IFesfDs  (please use the defintion from the Type Library) |
| CLSID | FesfDs (please use the definition from the Type Library) |

# UpdateHeader function

Updates the header of an FESF Encrypted file.

## Syntax

```
HRESULT
UpdateHeader(
    [in] REFGUID Volume,
    [in] BSTR    Path,
    [in] VARIANT *  NewHeader
   )
{ ... }
```

## Parameters

*Volume* [in]

A reference to a GUID that identifies the volume on which the file resides. See Remarks for further information.

*Path* [in]

A string containing the path of a file to update. This may be a fully qualified path or a path relative to the *Volume* argument. Refer to the Remarks section.

*NewHeader* [in]

A pointer to a VARIANT that describes a *CComSafeArray* of bytes holding the new header that is to be substituted for the existing header on the file.

## Return value

**S_OK** on success

The error code **E_INVALIDARG** is returned if the function is called with a header that is larger than can be accommodated in the existing file without extension.

Other standard HRESULT values may be returned indicating the failure of the operation.

## Remarks

A Client Solution component calls **UpdateHeader** to replace the existing Policy Header Data in an FESF encrypted file with new Policy Header Data. The old Policy Header Data is discarded.

If *Volume* is equal to FE_SHADOW_VOLUME_GUID (indicating that *Path* refers to a file on a network volume), then path is interpreted as *either* as a fully qualified path, suitable for direct evaluation *or* as the concatenation of the shadow device name and path.

If *Volume* is equal to GUID_NULL (indicating that no GUID is provided) then the path is inspected and treated as though either FE_SHADOW_VOLUME_GUID or FE_NETWORK_GUID was provided.

Otherwise, Volume represents a local volume and Path is interpreted as relative to that volume.

The new data may must be less than or equal to the number of bytes returned in the *MaxHeaderLength* parameter of the **FesfDS->ReadHeader** method. If the Client Solution component needs to write a header that's larger than *MaxHeaderLength* bytes, it must use the **FesfDS->UpdateHeaderWithExtension** method.

**UpdateHeader** is optimized to allow a Client Solution to update an existing header within specific size constraints, and with as little overhead as possible. If the size of the Header Data to be written is less than the maximum Header Data length returned in the *MaxHeaderLength* parameter of the **FesfDS->ReadHeader** function, the Client Solution component can call **FesfDS->UpdateHeader**. If the Header Data to be written is greater than *MaxHeaderLength* bytes, the **FesfDS->UpdateHeaderWithExtension** function must be used.

Solutions should note that, depending on the size of the new Header Data to be written, **UpdateHeader** may not inherently be transactionally safe. Thus, the file being updated could become corrupted if an unrecoverable error occurs during the header update process. If absolute safety is required, the Client Solution component should make a backup copy of the file being updated, call **UpdateHeader**, and when the update succeeds delete the backup copy.

The calling COM client must have SE_RESTORE_NAME privilege available to call this function.

## Example

```
//
// Now write the header back to the safe array so that we can
// write it out to the file
//
for (auto index = (ULONG)0; index < newLength; index++)
{

    headerAsSafeArray.SetAt(index, newHeader[index]);
}

if (action == HeaderUpdateSeqNumber || action == HeaderUpdateSizeRandom ||
        action == HeaderUpdateSizeIncreasing)
{
    //
    // If the header is larger than the maximum allowed size,
    // we have to call a different function to do it
    //
    if (newLength <= maxHeaderLength)
    {
```

```
        hr = spDS->UpdateHeader(const_cast<GUID *>(&GUID_NULL),
                                GetUniversalFilePath(file),
                                &variantHeader);
    }

    else
    {
        hr = spDS->UpdateHeaderWithExtension(
                    const_cast<GUID *>(&GUID_NULL),
                    GetUniversalFilePath(file),
                    &variantHeader);
    }

    if (!SUCCEEDED(hr))
    {
        wprintf(L"%-30ws <E> UpdateHeader failed (hr=0x%x)\n", findData.cFileName, hr);
        break;
    }
```

The above snippet comes from the SampUpdateHeader example that's provided as part of the UM_SAMPLE Solution. Refer to that example for more details.

## See also

FesfDS->UpdateHeaderWithExtension.

## Requirements

| | |
|---|---|
| Software version | FESF V1 (and later) |
| DLL/Server | FesfDs.exe |
| Supported FESF State | FESF Online State or FESF Offline State (as long as FesfDs in accessible via COM). |
| Type Library | \UM_FESF\UMLIB\FESFDS.TLB |
| IID | IFesfDs (please use the defintion from the Type Library) |
| CLSID | FesfDs (please use the definition from the Type Library) |

# UpdateHeaderWithExtension function

Updates the header of an FESF Encrypted file, extending the file to accommodate a larger Policy Header Data size.

## Syntax

```
HRESULT
UpdateHeaderWithExtension(
    [in] REFGUID Volume,
    [in] BSTR    Path,
    [in] VARIANT *  NewHeader
   )
{ ... }
```

## Parameters

*Volume* [in]

A reference to a GUID that identifies the volume on which the file resides.  See Remarks for further information.

*Path* [in]

A string containing the path of a file to update.  This may be a fully qualified path or a path relative to the *Volume* argument.  Refer to the Remarks section.

*NewHeader* [in]

A pointer to a VARIANT that describes a *CComSafeArray* of bytes holding the new header that is to be substituted for the existing header on the file.

## Return value

**S_OK** on success

Other standard HRESULT values may be returned indicating the failure of the operation.

## Remarks

A Client Solution component calls **UpdateHeaderWithExtension** to replace the existing Policy Header Data in an FESF encrypted file with new Policy Header Data, extending the file to accommodate the new Header Data.  The old Policy Header Data is discarded.

If *Volume* is equal to FE_SHADOW_VOLUME_GUID (indicating that *Path* refers to a file on a network volume), then path is interpreted as *either* as a fully qualified path, suitable for direct evaluation *or* as the concatenation of the shadow device name and path.

If *Volume* is equal to  GUID_NULL  (indicating that no GUID is provided) Then the path is inspected and treated as though either FE_SHADOW_VOLUME_GUID or FE_NETWORK_GUID was provided.

.  Otherwise, Volume represents a local volume and Path is interpreted as relative to that volume.

If the size of the Header Data to be written is less than or equal to the maximum Header Data length returned in the *MaxHeaderLength* parameter of the **FesfDS->ReadHeader** function, it is significantly less overhead for the Client Solution component to call **FesfDS->UpdateHeader** than to call **FesfDS->UpdateHeaderWithExtension**.

The new Header Data can actually be any size.  While the name of this function implies that the new Header Data will be larger than the existing Header Data (with the file being extended accordingly), the new Header Data may actually be smaller than the existing Header Data.  In this case, the file will be *shrunk* accordingly.

If the Client Solution component needs to write a header that's larger than *MaxHeaderLength* bytes, it must use the **FesfDS->UpdateHeaderWithExtension** method.

Solutions should note that, while **UpdateHeaderWithExtension** makes a reasonable attempt to ensure correctness, the function is not inherently be transactionally safe. Thus, the file being updated could become corrupted if an unrecoverable error occurs during the header update or file extension process.  If absolute safety is required, the Client Solution component should make a backup copy of the file being updated, call **UpdateHeaderWithExtension**, and when the update succeeds delete the backup copy.

The calling COM client must have SE_RESTORE_NAME, SE_SECURITY_NAME, and SE_TAKE_OWNERSHIP_NAME privileges available.

## Example

```
//
// Now write the header back to the safe array so that we can
// write it out to the file
//
for (auto index = (ULONG)0; index < newLength; index++)
{

    headerAsSafeArray.SetAt(index, newHeader[index]);
}

if (action == HeaderUpdateSeqNumber || action == HeaderUpdateSizeRandom ||
        action == HeaderUpdateSizeIncreasing)
{
    //
    // If the header is larger than the maximum allowed size,
    // we have to call a different function to do it
```

```
//
if (newLength <= maxHeaderLength)
{
    hr = spDS->UpdateHeader(const_cast<GUID *>(&GUID_NULL),
                            GetUniversalFilePath(file),
                            &variantHeader);
}

else
{
    hr = spDS->UpdateHeaderWithExtension(
                const_cast<GUID *>(&GUID_NULL),
                GetUniversalFilePath(file),
                &variantHeader);
}

if (!SUCCEEDED(hr))
{
    wprintf(L"%-30ws <E> UpdateHeader failed (hr=0x%x)\n", findData.cFileName, hr);
    break;
}
```

The above snippet comes from the SampUpdateHeader example that's provided as part of the UM_SAMPLE Solution. Refer to that example for more details.

## See also

FesfDS->UpdateHeader

## Requirements

| | |
|---|---|
| Software version | FESF V1 (and later) |
| DLL/Server | FesfDs.exe |
| Supported FESF State | FESF Online State or FESF Offline State (as long as FesfDs in accessible via COM). |
| Type Library | \UM_FESF\UMLIB\FESFDS.TLB |
| IID | IFesfDs (please use the defintion from the Type Library) |
| CLSID | FesfDs (please use the definition from the Type Library) |

# FESFSa Function Reference

The functions in this section are implemented by the FESF Stand-Alone library (FesfSa.lib).

## About the FESFSa Functions

A key point to keep in mind about the FesfSa functions is that they may be used only on systems where FESF is not installed (or where it can be unambiguously guaranteed that none of the FESF kernel-mode or user-mode components are running). This is referred to as FESF Not Installed state. Using the FesfSa functions on systems where the FESF is active will result in undefined behaviors, including file corruption.

Given the design patterns implemented by FesfSaEncrypt and FesfSaDecrypt, your stand-alone application's code will be responsible for performing the actual encryption and decryption operations on file data. This is in contrast to the practice when FESF is installed, in which FESF performs all encryption and decryption operations.

Because you will be responsible for implementing the data encryption and decryption functions, it should go without saying that, in order for newly encrypted files to be recognized and accessible by FESF, you must perform encryption and overall file operations in a way that is fully compatible with FESF. FESF uses the Microsoft CNG implementation for its encryption operations. Ensuring that your stand-alone application creates compatible FESF encrypted files is the responsibility of your application.

For algorithms requiring a fixed block size, we use a value of 256 bytes. This choice is arbitrary. Normally, algorithms that provide a CBC mode also include a non-secret value known as the initialization vector. This prevents identical blocks from appearing to be identical in the encrypted file content.

When calling CNG encryption methods that require an initialization vector (IV), FESF generates this from the key material using a technique adapted from the disk drive field and known as the Encrypted Salt-Sector Initialization Vector (ESSIV). In general:

$$IV(O)=C_S(O)$$

Where **O** is the offset, **C** is a cryptographic function and **s** is a cryptographic hash of the password. However, our specific implementation varies slightly. Specifically, we limit the value of **O** to be less than 263 bits (8EB) in keeping with the largest possible size in the Windows environment. Rather than sign extend the value, we shift the value by 64 bits. Thus, we implement:

$$IV(O)=C_S(O*2^{64})$$

We derive **s** (the encryption key we use) by computing the SHA-256 checksum of the file key material. This provides us with the "salt" to permute the underlying encryption algorithm, as necessary.

Our choice for C (the cryptographic function) is AES-256 in Electronic Code Book (ECB) mode.

# FesfSaDecrypt function

Decrypts a valid FESF encrypted file using a caller-provided callback.

## Syntax

```
bool
FesfSaDecrypt(
    _In_ const wchar_t *Path,
    _In_ FE_DECRYPT_CALLBACK_ROUTINE CallbackRoutine
    _In_ void *CallbackContext,
   )
{ ... }
```

## Parameters

*Path* [in]
A string containing the path of a file to decrypt. Refer to the Remarks section.

*CallbackRoutine* [in]
A pointer to a caller supplied *DecryptCallback* routine. Refer to the Remarks section.

*CallbackContext* [in]
A pointer to a caller-provided context structure passed to every invocation of the provided WriteCallback.

## Return value

Returns **true** if the function successfully processes all the data, **false** otherwise.

For Windows platforms, a specific status code for this function is reported with GetLastError().

For Linux platforms, the errno variable is set.

## Remarks

This function provides FESF with a path describing a file to be decrypted. FESF calls the application provided *Callback* function with data blocks from the file for the callback to decrypt. FESF does not call the Callback function with any metadata (FESF metadata or the Solution Header Data).

The file described by Path must be a valid FESF encrypted file. If it is not, an error is returned. If FESF cannot open the file described by Path for exclusive access, an error is returned.

The callback is called synchronously with respect to this function.  That is, the application's call to Decrypt returns when all data has been supplied by FESF to the callback.

## Example

## See also

## Requirements

| | |
|---|---|
| Software version | FESF Version 1 (added) |
| Supported FESF State | FESF Not Installed ONLY |
| Windows Library | FesfSa.lib |
| Linux Library | FESFsa.a |

# DecryptCallback routine

Receives a block of data read from a FESF encrypted file for decryption.

## Syntax

```
FE_DECRYPT_CALLBACK_FUNCTION DecryptCallback;

bool
DecryptCallback(
    _In_ void *CallbackContext,
    _In_ void *SolutionHeader,
    _In_ uint32_t SolutionHeaderSize,
    _In_ uint64_t FinalSize,
    _In_ void *EncryptedData,
    _In_ uint32_t EncryptedDataSize
   )
{ ... }
```

## Parameters

*CallbackContext* [in]
A buffer containing context data provided to the Decrypt function.

*SolutionHeader* [in]
A buffer that contains the Solution Header retrieved by FESF from the encrypted file.

*SolutionHeaderSize* [in]
The size of the buffer pointed to by SolutionHeader, in bytes.

*FinalSize* [in]
The final size of the decrypted output for the file. Does not change across a single invocation of the Decrypt function that calls this callback.

*EncryptedData* [in]
A buffer containing the next block of encrypted data.

*EncryptedDataSize* [in]
The length of the data to be decrypted and written. Always provided as a multiple of CipherBlockSize, even if the decrypted contents may be of a different size. See Remarks.

## Return value

Returns **true** if the function successfully processes all the data, **false** otherwise.

For Windows platforms, a specific status code for this function is reported with SetLastError().

For Linux platforms, the errno variable should be set to report a specific error code.

## Remarks

This callback routine is called to provide encrypted data to an application to allow the application to produce a data stream that contains unencrypted data.

FESF reads encrypted data blocks from the file and provides them sequentially to the application via this callback. The callback decrypts the data provided.

The last block in a file may contain data padding as required by some encryption algorithms. Care should be taken to not write more data to the output stream than is specified by the *FinalSize* parameter.

If the encryption algorithm requires an Initialization Vector (IV), the application is required to use the same algorithm that FESF uses to generate a unique IV per cipher block. For chained ciphers such as CBC, the encryption algorithm is likewise required to implement the same blocking scheme used by FESF. See the section **About the FESFSa Functions** in this document for the description of these issues.

## Example

## See also

## Requirements

| | |
|---|---|
| Software version | FESF Version 1 (added) |
| Supported FESF State | FESF Not Installed ONLY |
| Windows Library | FesfSa.lib |
| Linux Library | FESFsa.a |

# FesfSaEncrypt function

Enables an application to create an FESF encrypted data stream (a file, a series of network messages, etc) from a plaintext file.

## Syntax

```
bool
FesfSaEncrypt(
    _In_ const wchar_t *Path,
    _In_ FE_ENCRYPT_WRITER CallbackRoutine
    _In_ void *CallbackContext,
    _In_ void *SolutionHeader,
    _In_ uint32_t SolutionHeaderSize,
    _In_ uint32_t CipherBlockSize,
   )
{ ... }
```

## Parameters

*Path* [in]

A string containing the path of a file to encrypt. Refer to the Remarks section.

*CallbackRoutine* [in]

A pointer to a caller supplied *EncryptCallback* routine. Encrypt calls this function for each segment of data in the FESF encrypted data stream.

*CallbackContext* [in]

A pointer to a caller-provided context structure passed to every invocation of the provided callback.

*SolutionHeader* [in]

A buffer that contains the Solution Header that FESF will include in its metadata in the process of encrypting the file.

*SolutionHeaderSize* [in]

The size of the buffer pointed to by SolutionHeader, in bytes.

*CipherBlockSize* [in]

The block size of the encryption algorithm used by the Callback function.

## Return value

Returns **true** if the function successfully processes all the data, **false** otherwise.

For Windows platforms, a specific status code for this function is reported with GetLastError().

For Linux platforms, the errno variable is set.

## Remarks

This function provides the caller-supplied *EncryptCallback* with a stream of sequential data that will produce a valid FESF encrypted file. The Callback will be called multiple times until all file data has been supplied. See the description of *EncryptCallback* for more information.

If FESF cannot open the file described by Path for exclusive access, an error is returned.

The provided *SolutionHeader* is identical to the *PolHeaderData* buffer returned by the Solution's Policy DLL from FESF's PolGetKeyNewFile callback.

The callback is called synchronously with respect to this function. That is, the application's call to Encrypt returns when all data has been supplied by FESF to the callback.

## Example

## See also

## Requirements

| | |
|---|---|
| Software version | FESF Version 1 (added) |
| Supported FESF State | FESF Not Installed ONLY |
| Windows Library | FesfSa.lib |
| Linux Library | FESFsa.a |

# EncryptCallback routine

Processes a block of data generated by the Encrypt call to encrypt and store a sequential output stream.

## Syntax

```
FE_ENCRYPT_CALLBACK_ROUTINE EncryptCallback;

bool
EncryptCallback(
    _In_ void *CallbackContext,
    _In_ uint64_t FinalSize,
    _In_ void *StreamData,
    _In_ uint32_t StreamDataLength,
    _In_ bool EncryptBeforeWriting
)
{ ... }
```

## Parameters

*CallbackContext* [in]
: A buffer containing context data provided to the Encrypt function.

*FinalSize* [in]
: The final size of the encrypted data stream. The resultant output must be exactly this size to be successfully recognized by FESF. See Remarks for more details.

*StreamData* [in]
: A buffer containing the unencrypted data to be written. If WriteEncrypted is **true**, the application receiving the callback is responsible for encrypting the contents of the data buffer.

*StreamDataLength* [in]
: The length of the data to be written. If WriteEncrypted is **true**, this will be a multiple of the CipherBlockSize argument passed to Encrypt.

*EncryptBeforeWriting* [in]
: If **true**, the contents of the Data buffer must be encrypted before storing the output. If **false**, Data must be written without modification.

## Return value

Returns **true** if the function successfully processes all the data, **false** otherwise.

For Windows platforms, a specific status code for this function can be reported with SetLastError().

For Linux platforms, the errno variable should be set to report a specific error code.

## Remarks

This callback routine is called to provide data to an application to allow the application to produce a data stream that can be interpreted as a valid FESF encrypted file.

If *EncryptBeforeWriting* is TRUE, the application receiving the callback is responsible for encrypting the supplied Data using key material that can be derived from the *SolutionHeader* it previously passed to the Encrypt function. If *EncryptBeforeWriting* is TRUE and the data supplied in Data is not an integer multiple of the *CipherBlockSize* the application receiving the callback is responsible for padding the data appopriately before performing the encryption operation.

If *EncryptBeforeWriting* is FALSE, the data supplied in the *Data* buffer is FESF metadata that must be stored exactly as supplied from the callback, without any change. These data blocks may not be padded or rounded in size.

Each block of callback data provided to this routine must appear contiguously and the same order in the output stream as it is provided to the callback.

If the encryption algorithm requires an Initialization Vector (IV), the application is required to use the same algorithm that FESF uses to generate a unique IV per cipher block. For chained ciphers such as CBC, the encryption algorithm is likewise required to implement the same blocking scheme used by FESF. See the section **About the FESFSa Functions** in this document for the description of these issues.

The *FinalSize* argument defines the ultimate size of the stream, which may be slightly larger than the number of data bytes written to the stream. This argument will be the same each time *Callback* is called for a given call to Encrypt.

Failure to write the data in the correct order, or failure to make sure that the file is exactly *FinalSize* bytes long will result in an inconsistent or invalid FESF encrypted file.

## Example

## See also

## Requirements

| | |
|---|---|
| Software version | FESF Version 1 (added) |
| Supported FESF State | FESF Not Installed ONLY |

| | |
|---|---|
| Windows Library | FesfSa.lib |
| Linux Library | FESFsa.a |

# FesfSaIsFileEncrypted function

Determines if a given file is stored in FESF encrypted format.

## Syntax

```
bool
FesfSaIsFileEncrypted(
    _In_const wchar_t *Path,
    _Out_ bool        * Encrypted
)
{ ... }
```

## Parameters

*Path* [in]
A string containing the path of a file to check. This must be a fully qualified path.

*Encrypted* [out, retval]
A pointer to a bool that will receive the result on success. Set to *true* if the file indicated by *Path* is in FESF encrypted format.

## Return value

Returns **true** if the indicated file is recognized as being encrypted by FESF, **false** otherwise.

For Windows platforms, a specific status code for this function is reported with SetLastError().

For Linux platforms, the errno variable should be set to report a specific error code.

## Remarks

*Path* is interpreted as a fully qualified path, suitable for direct evaluation.

## Example

## See also

## Requirements

| | |
|---|---|
| Software version | FESF Version 1 (added) |

| | |
|---|---|
| Supported FESF State | FESF Not Installed ONLY |
| Windows Library | FesfSa.lib |
| Linux Library | FESFsa.a |

# FesfSaReadHeader function

Reads the Application Header.

## Syntax

```
bool
FesfSaReadHeader(_In_ const wchar_t *Path,
                 _Inout_opt_bytecount_( SolutionHeaderSize) void * SolutionHeader,
                 _In_  uint32_t SolutionHeaderSize,
                 _Out_ uint32_t *BytesRead
);)
{ ... }
```

## Parameters

*Path* [in]
A string containing the path of a file whose solution header should be read.

*SolutionHeader* [out, retval]
A caller allocated buffer to receive the solution header.

*SolutionHeaderSize* [in]
The size of the caller allocated buffer in bytes.

*BytesRead* [out]
A pointer to an integer which will receive the size of the solution header.

## Return value

Returns **true** if the header was successfully read, **false** otherwise.

For Windows platforms, a specific status code for this function is reported with SetLastError().

For Linux platforms, the errno variable should be set to report a specific error code.

## Remarks

*Path* is interpreted as a fully qualified path, suitable for direct evaluation.

If the supplied buffer is too small then false is returned, but BytesRead is set to the size of the solution header in the file. Additionally in this situation, for Windows platforms GetLastError is set to be `ERROR_BUFFER_OVERFLOW`, and for non-Windows platforms errno is set to be `-E2BIG`.

In all other error cases BytesRead is set to be `0XFFFFFF.`

## Requirements

| | |
|---|---|
| Software version | FESF Version 1.1 (added) |
| Supported FESF State | FESF Not Installed ONLY |
| Windows Library | FesfSa.lib |
| Linux Library | FESFsa.a |

# FesfSaWriteHeader function

Writes the Application Header.

## Syntax

```
bool
FesfSaWriteHeader(_In_ const wchar_t *Path,
                  _In_ void *SolutionHeader,
                  _In_ uint32_t SolutionHeaderSize)
{ ... }
```

## Parameters

*Path* [in]
A string containing the path of a file whose solution header should be written.

*SolutionHeader* [in]
A buffer containing the header

*SolutionHeaderSize* [in]
The size of the buffer in bytes.

## Return value

Returns **true** if the header was successfully written, **false** otherwise.

For Windows platforms, a specific status code for this function is reported with SetLastError().

For Linux platforms, the errno variable should be set to report a specific error code.

## Remarks

*Path* is interpreted as a fully qualified path, suitable for direct evaluation.

If the new Solution Header is larger than the current one and there is no room to accommodate it in the file then this call fails.

## Requirements

| | |
|---|---|
| Software version | FESF Version 1.1 (added) |

| | |
|---|---|
| Supported FESF State | FESF Not Installed ONLY |
| Windows Library | FesfSa.lib |
| Linux Library | FESFsa.a |

# FESF Policy Data Structures

# FE_POLICY_CONFIG structure

The FE_POLICY_CONFIG structure specifies the selected configuration options and callbacks for the Policy DLL.

## Syntax

```
typedef struct _FE_POLICY_CONFIG {

    DWORD                           VersionMajor;
    DWORD                           VersionMinor;
    DWORD                           Length;

    struct {
        bool  ApproveRename;
        bool  ApproveCreateLink;
        bool  ApproveCorruptFileAccess;
        bool  RawDirSize;
    } OfflineBehavior;

    struct {
        bool  Enable;
    } AccessCache;

    POL_GET_POLICY_NEW_FILE       *PolGetPolicyNewFile;
    POL_GET_KEY_NEW_FILE          *PolGetKeyNewFile;
    POL_GET_POLICY_EXISTING_FILE  *PolGetPolicyExistingFile;
    POL_GET_KEY_FROM_HEADER       *PolGetKeyFromHeader;

    POL_APPROVE_RENAME            *PolApproveRename;
    POL_APPROVE_CREATE_LINK       *PolApproveCreateLink;
    POL_REPORT_FILE_INCONSISTENT  *PolReportFileInconsistent;
    POL_REPORT_LAST_HANDLE_CLOSED *PolReportLastHandleClosed;
    POL_FREE_HEADER               *PolFreeHeader;
    POL_FREE_KEY                  *PolFreeKey;

    POL_UNINIT                    *PolUnInit;

    DWORD                         AlgorithmsCount;
    FE_POLICY_ALGORITHM           *Algorithms[1];

} FE_POLICY_CONFIG;
```

## Members

**VersionMajor**

The major version of the FESF Policy API supported by the Policy DLL. This must be **FE_POLICY_VERSION_MAJOR**.

**VersionMinor**

The minor version of the FESF Policy API supported by the Policy DLL. This must be **FE_POLICY_VERSION_MINOR**.

**Length**

The length in bytes of the FE_POLICY_CONFIG structure.

**OfflineBehavior**

The fields in this structure set the default values that FESF kernel mode components should use when the FESF Policy Service is not running (that is, FESF is running on Offline State). This state can occur (a) after the kernel mode components have started and before FesfPolicy has started, (b) FesfPolicy fails or/or is being restarted, or (c) during system shutdown, after FesfPolicy has terminated but before the system has completed shutdown processing.

The behaviors specified in this section are saved in the Registry, and used by FESF during subsequent reboot operations.

**ApproveRename**

If set to **true** rename operations will be allowed if the FESF Policy Service is not running.

**ApproveCreateLink**

If set to **true** create hard link operations will be allowed if the FESF Policy Service is not running.

**ApproveCorruptFileAccess**

If set to **true,** access to files that are in FESF format but that are "inconsistent" will be allowed when the FESF Policy Service is not running. Files that are "inconsistent" are those which FESF identifies as have an internal structure issue. See the description of *PolReportFileInconsistent* for more details.

**RawDirSize**

If set to true, the file sizes shown by directory enumeration will reflect what is consumed on disk (allowing for the Solution Header). The default is to show size of the data in the file. See PolGetPolicyDirectoryListing for how to control this behavior while the service is operating.

**AccessCache**

**Enable**

Set to **true** to enable FESF Policy Caching. Otherwise, set to **false**.

**PolGetPolicyNewFile**

A pointer to the Client Solution Policy DLL's *PolGetPolicyNewFile* callback function.

**PolGetKeyNewFile**

A pointer to the Client Solution Policy DLL's *PolGetKeyNewFile* callback function.

**PolGetPolicyExistingFile**

A pointer to the Client Solution Policy DLL's *PolGetPolicyExistingFile* callback function.

**PolGetKeyFromHeader**
A pointer to the Client Solution Policy DLL's *PolGetKeyFromHeader* callback function.

**PolApproveRename**
A pointer to the Client Solution Policy DLL's *PolApproveRename* callback function.

**PolApproveCreateLink**
A pointer to the Client Solution Policy DLL's *PolApproveCreateLink* callback function.

**PolReportFileInconsistent**
A pointer to the Client Solution Policy DLL's *PolReportFileInconsistent* callback function.

**PolReportLastHandleClosed**
A pointer to the Client Solution Policy DLL's *PolReportLastHandleClosed* callback function.

**PolFreeHeader**
A pointer to the Client Solution Policy DLL's *PolFreeHeader* callback function.

**PolFreeKey**
A pointer to the Client Solution Policy DLL's *PolFreeKey* callback function.

**PolUnInit**
A pointer to the Client Solution Policy DLL's *PolUnInit* callback function.

**AlgorithmsCount**
A count of entries in the vector at the **Algorithms** member of this structure.

**Algorithms**
Pointer to a vector of **FE_POLICY_ALGORITHM** structures, each of which describes an encryption algorithm that the Policy DLL will use.

## Remarks

## See Also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Header | PolDllApi.h |

# FE_POLICY_PATH_INFORMATION structure

The FE_POLICY_PATH_INFORMATION structure specifies the path name for a file being accessed by FESF.

## Syntax

```
typedef struct _FE_POLICY_PATH_INFORMATION {
    LPCWSTR RelativePath;
    DWORD   PathFlags;
    UUID    VolumeGuid;
    Union {
          LPCWSTR ServerAndShare;
          LPCWSTR ShadowVolumeName;
          };
} FE_POLICY_PATH_INFORMATION;
```

## Members

**RelativePath**

A path name (including file name), starting with backslash. For local volumes, the path is relative to the volume GUID. For network volumes, the path is relative to the share.

**PathFlags**

A bitmask containing values describing the location of the path being provided:

| | |
|---|---|
| **FE_POLICY_PATH_NAME_NOT_NORMALIZED** 0x001 | The path information provided has NOT been normalized in format. This is a rare occurrence and relates only to some specific network operations. |

**VolumeGuid**

For network files, this field contains the GUID FE_NETWORK_GUID.

For shadow volumes, this field contains the GUID FE_SHADOW_VOLUME_GUID.

For local (that is, non-network) volumes, this field contains the GUID representing the local volume on which the file resides. The drive letter that this GUID represents can be translated and combined with the contents of the *RelativePath* field using the FESF Utility Library function **GetFullyQualifiedLocalPath.** The result will be a traditional Windows fully qualified path name.

**ServerAndShare**

The name of the server and share. The UNC file name can be derived by appending the **RelativePath** to the **ServerAndShare.**

Only valid if **VolumeGuid** is FE_NETWORK_GUID.

**ShadowVolumeName**

The “device name” of the shadow volume. A UNC file name can be derived by appending **RelativePath** to the **ShadowVolumeName** and prepending the whole with `\\?\GlobalRoot`

Only valid if **VolumeGuid** is FE_SHADOW_VOLUME_GUID.

## Remarks

## See Also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Header | PolDllApi.h |

# FE_POLICY_ALGORITHM_PROPERTY structure

The FE_POLICY_ALGORITHM_PROPERTY structure specifies an encryption algorithm property to be passed by FESF to CNG.

## Syntax

```
typedef struct _FE_POLICY_ALGORITHM_PROPERTY {
    LPCWSTR CNGPropertyIdentifier;
    PVOID   CNGPropertyValue;
    DWORD   CNGPropertyValueLength;
} FE_POLICY_ALGORITHM_PROPERTY;
```

## Members

**CNGPropertyIdentifier**

A pointer to a constant null-terminated wide-character string that contains the name of a property. Strings containing standard cryptographic primitive property identifiers defined by CNG (such as BCRYPT_CHAINING_MODE, BCRYPT_INITIALIZATION_VECTOR, etc.) are defined in the standard Windows header file **bcrypt.h**. The string provided here by the Policy DLL will be provided by FESF to CNG as the *pszProperty* argument on a call to **BCryptSetProperty**. See the MSDN documentation for that function for more information.

**CNGPropertyValue**

A pointer to an untyped buffer containing the value for the property. The **CNGPropertyValueLength** member contains the length of this buffer. This buffer is provided by FESF as input to CNG as the *pbInput* argument on a call to **BCryptSetProperty**.

**CNGPropertyValueLength**

The length of the buffer pointed to by **CNGPropertyValue**.

## Remarks

## See Also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (or later) |
| Header | PolDllApi.h |

# FE_POLICY_ALGORITHM structure

The FE_POLICY_ALGORITHM structure specifies the encryption algorithm and associated information. This information is used by FESF to call CNG.

## Syntax

```
typedef struct _FE_POLICY_ALGORITHM {
    LPCWSTR                       PolUniqueAlgorithmId;
    LPCWSTR                       CNGAlgorithmIdentifier;
    LPCWSTR                       CNGAlgorithmImplementation;
    DWORD                         PropertiesCount;
    FE_POLICY_ALGORITHM_PROPERTY  Properties[1];
} FE_POLICY_ALGORITHM;
```

## Members

**PolUniqueAlgorithmId**

A pointer to a Policy DLL defined null-terminated constant wide character string, that will be used to identify this particular algorithm and specified properties. The Policy DLL provides this string as an output from its *PolGetKeyNewFile* and *PolGetKeyFromHeader* callback functions.

**CNGAlgorithmIdentifier**

A pointer to a null-terminated, constant, wide character string that will be used by FESF to identify the requested cryptographic algorithm to CNG. Strings for standard CNG algorithms (such as BCRYPT_AES_ALGORITHM, BCRYPT_3DES_ALGORITHM, etc.) are defined in the standard Windows header file **bcrypt.h**. The string provided here by the Policy DLL will be provided to CNG as the *pszAlgId* argument when FESF calls **BCryptOpenAlgorithmProvider**. See the MSDN documentation for that function for more information.

**CNGAlgorithmImplementation**

A pointer to a null-terminated, constant, wide character string that identifies the specific cryptographic algorithm provider to load. This parameter is optional and is typically NULL. FESF provides this value as the *pszImplementation* argument when it calls **BCryptOpenAlgorithmProvider**. See the MSDN documentation for that function for more information.

**PropertiesCount**

A count of the number of properties provided in the **Properties** member of this structure.

**Properties**

A vector of **FE_POLICY_ALGORITHM_PROPERTY** structures, each of which defines a specific algorithm property.

## Remarks

## See Also

## Requirements

| | |
|---|---|
| Software version | FESF V1 (and later) |
| Header | PolDllApi.h |