

OSR FILE ENCRYPTION SOLUTION FRAMEWORK

# SOLUTION DEVELOPERS' GUIDE

## V2.0

## Table of Contents

1	Copyright Notice .....	6
2	Change Log .....	7
3	Introduction .....	8
4	FESF Overview and Basic Concepts .....	9
4.1	Policy .....	9
4.2	Policy Definition and Storage .....	10
4.3	Communicating Policy from Client Solution to FESF .....	11
4.4	Key Material and Encryption Identification .....	11
4.5	Where is the Encryption Actually Done and What Algorithms Are Supported? .....	11
4.6	Existing Files Are Not Automatically Encrypted .....	12
4.7	The Basics of Policy Operation.....	12
4.8	Exempting Drives/Shares from Policy Determinations .....	13
4.9	FESF Policy Caching .....	14
4.10	Files or Streams? .....	14
5	FESF Components and Interfaces.....	16
5.1	FESF Kernel Mode Components .....	17
5.2	FESF User Mode Components .....	17
5.3	Client Solution Components .....	18
6	Designing and Building a Solution.....	20
6.1	Solution Policy DLL Callback Functions .....	20
6.2	FESF Policy Before the Solution Policy DLL Starts .....	22
6.3	Solution Policy DLL Initialization.....	22
6.4	Returning Failure from Solution Policy DLL Callbacks .....	22
6.5	Guidance for Implementing Callback Functions.....	22
6.6	Guidance Regarding Solution Policy DLL Solution Header Data .....	24
6.7	Working with Local, Network, and Shadow Volume File Paths .....	24
6.8	A Note About Raw File Access.....	25
6.9	Installing and Using Fe2Policy .....	26
6.10	FESF Kernel Component Logging and Tracing .....	30
7	Building FESF From Source .....	32

7.1	FESF Canonical User-Mode Components.....	32
7.2	FESF Sample Solution Components .....	32
7.3	FESF Kernel Mode Components .....	33
8	Notes on Installing FESF with Your Solution.....	35
8.1	About the Sample Installer Project.....	35
8.2	About Customizing FESF Component Names .....	35
9	FESF Known Restrictions and Limitations.....	37
10	Solution Policy DLL Callback Function Reference.....	39
10.1	About Implementing Your Callback DLL .....	39
	PolicyDllInit callback function.....	41
	PolApproveCreateLink callback function.....	43
	PolApproveRename callback function.....	45
	PolApproveTransactedOpen callback function .....	47
	PolAttachVolume callback function .....	49
	PolFreeHeader callback function.....	51
	PolFreeKey callback function .....	52
	PolGetKeyFromHeader callback function.....	54
	PolGetKeyNewFile callback function.....	57
	PolGetLockRounding callback function.....	60
	PolGetPolicyDirectoryListing callback function .....	62
	PolGetPolicyExistingFile callback function .....	64
	PolGetPolicyNewFile callback function.....	67
	PolReportFileInconsistent callback function .....	71
	PolReportLastHandleClosed callback function.....	72
	PolUnInit callback function .....	74
11	FESF Policy Function Reference .....	75
	FePolSetConfiguration function .....	76
12	FesfUtil2 Function Reference.....	77
12.1	Using FesfUtil2 .....	77
	FesfUtil2FixFileTag Function.....	80
	FesfUtil2GetExecutablePathForThreadId Function.....	81

FesfUtil2GetFileSize Function.....	82
FesfUtil2GetFullyQualifiedPath Function.....	83
FesfUtil2GetSidForThreadId Function (deprecated).....	84
FesfUtil2GetUniversalFilePath Function.....	85
FesfUtil2GetVersion Function .....	86
FesfUtil2IsFileFesfEncrypted Function.....	87
FesfUtil2IsThreadIdInSid Function.....	88
FesfUtil2PurgePolicyCache Function (file variant).....	89
FesfUtil2PurgePolicyCache Function (thread variant).....	90
FesfUtil2ReadHeaderExclusive Function.....	91
FesfUtil2ReadHeaderUnsafe Function .....	92
FesfUtil2SetPolicyCacheState Function .....	93
FesfUtil2UpdateHeaderExclusive Function.....	94
FesfUtil2UpdateHeaderExclusiveWithExtension Function.....	95
FesfUtil2UpdateHeaderUnsafe Function .....	97
12.2 FesfUtil2 Classes & Structures.....	98
FESF_UTIL2_SOLUTION_HEADER.....	99
FEU2Exception .....	101
13 FESF Stand-Alone Library Function Reference.....	102
13.1 About the Stand-Alone Library.....	102
13.2 About the FE2Sa Functions.....	102
DecryptCallback function .....	104
EncryptCallback function.....	106
FesfSaDecrypt function.....	108
FesfSaEncrypt function .....	110
FesfSaIsFileEncrypted function .....	112
FesfSaReadHeader function.....	113
FesfSaWriteHeader function .....	115
14 FESF Policy Data Structures .....	116
FE2_POLICY_ALGORITHM structure .....	117
FE2_POLICY_CONFIG structure.....	120

FE\_POLICY\_PATH\_INFORMATION structure .....125

FE\_POLICY\_VOLUME\_INFORMATION structure .....127

15 FESF and Support for Cloud Storage .....129

15.1 Why Interoperability Is Complex .....129

15.2 OSR’s Approach to Interoperability for FESF .....132

15.3 FESF Interoperability with Cloud Storage Products .....134

15.4 Cloud Storage Products We Test With .....135

15.5 Summary and OSR’s Recommendations .....136

# 1 Copyright Notice

---

## © 2015-2025 OSR Open Systems Resources, Inc.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means -- graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems -- without written permission of OSR Open Systems Resources, Inc., 889 Elm St, 6<sup>th</sup> Floor, Manchester, New Hampshire 03101 USA | (603) 595-6500 | info@osr.com

OSR, the OSR logo, "OSR Open Systems Resources, Inc.", and "The NT Insider" are registered trademarks of OSR Open Systems Resources, Inc. All other trademarks mentioned herein are the property of their owners.

## U.S. GOVERNMENT RESTRICTED RIGHTS

This material is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Right in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software--Restricted Rights 48 CFR 52.227-19, as applicable. Manufacturer is OSR Open Systems Resources, Inc. Manchester, New Hampshire 03101.

## 2 Change Log

Document Date	FESF Version	Location of Change	Description of Change
1 April 2025	V2.0 Beta 1		Revisions to reflect changes in FESF V2.
August 2025	V2.0 Release		Updates for Release, particularly in AttachVolume, Fe2Sa, installation, and others.

## 3 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" (or, alternatively, "licensee") 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 make 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.



## 4 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, it turns out to be something that's exceptionally complicated. Creating a Solution that provides on-access encryption that performs well is even more difficult.

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

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

### 4.1 Policy

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

- Whether data written to a newly created file should be transparently encrypted before being stored on disk. Transparent encryption also implies storing Client-defined control information and FESF metadata information within the file.
- 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 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 user 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 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.
  - 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.
- The Thread ID (TID) of the thread that is creating or accessing the file. Given this TID, the Solution can determine the fully qualified path of the executing image, using an FESF-provided utility function.
- 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 user or security group) that is performing the file access. See <https://learn.microsoft.com/en-us/windows/win32/secauthz/security-identifiers> for more information.

- The file access (read data, write data, and others) that was granted to the accessing application.
- The action taken on the file being accessed (technically, the file's "disposition"). This indicates whether the file is being newly created, overwritten, appended, or just opened.

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

- Volume (or server and share), directory path, and name of the file being accessed
- Path and name of the accessing application
- Security ID 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 considered, and how they may be used to define Policy, is entirely up to the Client Solution. In addition, variables other than those above 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 Solution might be:

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

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

"Decrypt all FESF 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.

## 4.2 Policy Definition and Storage

So, where and how is Policy determined? And, once Policy has been determined 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.

### 4.3 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 (Fe2Policy). Fe2Policy is a standard Windows user-mode service that runs under the local system account. The callback function that Fe2Policy calls is provided by the Solution in a DLL known as the Solution Policy DLL. Fe2Policy loads this DLL dynamically when it starts based on a Registry parameter.

We'll describe a great deal more about the Solution Policy DLL later in this document. However, what's important to understand at this point is that the Solution Policy DLL is the (one and only) way that FESF asks the Client Solution for Policy decisions. Thus, the Solution 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, Fe2Policy will call the Solution Policy DLL's *PolGetPolicyNewFile* callback function. As the return value from this function, the Solution Policy DLL indicates whether data should be encrypted when written to the file that's being created or data should be written to the file that's being created as clear text. Similarly, each time an existing FESF encrypted file is opened, the Solution 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 to the ciphertext without transparent encryption or decryption).

### 4.4 Key Material and Encryption Identification

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

1. **Solution Header Data:** This data – which is entirely defined by the Client Solution – will be stored by FESF in the newly created file exactly as provided by the Client Solution. This Solution Header Data will be provided by FESF to the Client Solution whenever the file is subsequently opened and the key is required. The Solution 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 Solution Policy DLL is called with the path of the file being opened and the Solution Header Data that was previously stored in the file (along with other information). This Solution Header Data was supplied by the Solution when the file was created. Using this Solution Header Data, the Solution Policy DLL 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.

### 4.5 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.

FESF has built-in support for using CNG to perform AES 128 CBC and AES 256 CBC encryption. Custom CNG Cryptographic Algorithm Providers can be written by Clients to support other algorithms supported by CNG, or even custom-built algorithms.

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.

On systems where FESF is not installed, some encryption and decryption may be performed by the Client Solution in user-mode with the assistance of FESF supplied Stand-Alone library functions and a user-supplied cryptographic implementation.

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

FESF uses a block-size value of 256 bytes. This choice is arbitrary, but not changeable. Algorithms that provide a CBC mode typically include a non-secret value known as the *initialization vector*. This prevents identical blocks from appearing to be identical in the encrypted file content.

For AES CBC methods, FESF generates the Initialization Vector (IV) from the key material using a technique known as the Encrypted Salt-Sector Initialization Vector (ESSIV). More details about how FESF generates the IV can be found in the FE2Sa Function Reference section, elsewhere in this document.

## 4.6 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 FESF 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 examples where we had the Policy:

"We want any files that are created in the directory \MySecretStuff\ on the volume that's the D 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 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.

It is also important to note that FESF does not provide any mechanism to directly encrypt an existing file. In fact, the easiest way for a Client Solution to "encrypt an existing file" is for the Solution to copy the existing (cleartext) file to a new file that its Solution Policy DLL decides should be encrypted, and then delete the original (cleartext) file.

## 4.7 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 running.

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

For each new file that's created, FESF calls the Solution Policy DLL at its *PolGetPolicyNewFile* callback function to determine the Policy for that file. In other words, FESF calls the Solution Policy DLL to determine whether the data written to the file should be encrypted. If the Solution Policy DLL indicates that the data should be encrypted, FESF next calls the Solution Policy DLL's *PolGetKeyNewFile* to get the Solution Header Data, Algorithm ID, and data encryption Key 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 its own control and consistency metadata information to the file, including the Client-defined Solution Header Data, to enable later validation and decryption.

If the Solution 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 Solution Policy DLL's *PolGetKeyNewFile* is not called, and FESF adds no additional information to this file.

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

For each existing FESF encrypted file that's accessed, the Solution 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 Policy DLL at its *PolGetKeyFromHeader* callback function. When FESF calls *PolGetKeyFromHeader*, FESF passes the Solution Header Data that was stored with the file and previously returned by the Solution Policy DLL's *PolGetKeyNewFile* callback when the file was created. Given this Solution Header Data (and any other data that it may collect), the Solution Policy DLL returns the Algorithm ID and the file's data encryption/decryption 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 stored in encrypted form.

### 4.8 Exempting Drives/Shares from Policy Determinations

FESF allows the Client Solution to determine which volumes and network shares are eligible to host FESF encrypted files. When each disk volume or network share is first discovered by Windows, FESF calls the Solution Policy DLL at its *AttachVolume* callback function to determine if that volume should be under FESF control. If the files on that volume will never need to be encrypted or decrypted by FESF, the Solution Policy DLL can instruct FESF to completely ignore the volume, thereby removing FESF from any file processing in the volume's path. When a volume is ignored the Solution Policy DLL will not be called (a) when a new file is created on that volume, (b) an existing FESF encrypted file is opened on the volume. When an existing FESF encrypted file is accessed on a volume that is not under FESF control, raw (encrypted) data is always returned.

Calls to the Solution Policy DLL's *AttachVolume* callback for volumes that are discovered by Windows during startup (boot) processing are deferred. These callbacks take place as a result of the first file access that occurs following the startup of Fe2Policy and the successful loading of the Solution Policy DLL.

The *AttachVolume* callback is a very powerful tool for Solution developers. It enables them to significantly reduce the number of callbacks that occur to their Solution Policy DLLs for volumes where Policy will never need to be applied.

## 4.9 FESF Policy Caching

A powerful feature of FESF is FESF Policy Caching. A Solution Policy DLL may enable FESF Policy Caching by setting the *AccessCache.Enable* field of the **FE2\_POLICY\_CONFIG** structure to **TRUE**. To ensure good system performance, OSR strongly recommends that 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 Solution 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 Solution 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 Solution Policy DLL. This eliminates the overhead of calling the Solution Policy DLL to determine policy for a file, process, and access type when the Solution 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) cache information for the file, whichever is longer. On systems with lots of free memory, cache information can persist for a very long time (many hours or even days) 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 Solution. The Solution can remove entries in the FESF Policy Cache for a given file or process by calling the FesUtil2 function **FesUtil2PurgePolicyCache**. This function can also be used to remove all FESF Policy Cache entries for all files for all processes. Refer to the docs for **FesUtil2PurgePolicyCache** for specific information.

Also, overall FESF Policy caching behavior can also be dynamically enabled or disabled by a Solution at any time. This is done using the FesUtil2 function **FesUtil2SetPolicyCacheState**. Refer to the docs for **FesUtil2SetPolicyCacheState** for more information.

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

## 4.10 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 Solution 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 Solution Policy DLL to establish Policy on a per stream basis and not just on a per file basis. Also, while FESF will not call the Solution Policy DLL for directories as a general rule, for file systems that support streams on directories it **will** call the Solution Policy DLL for streams created on directories.

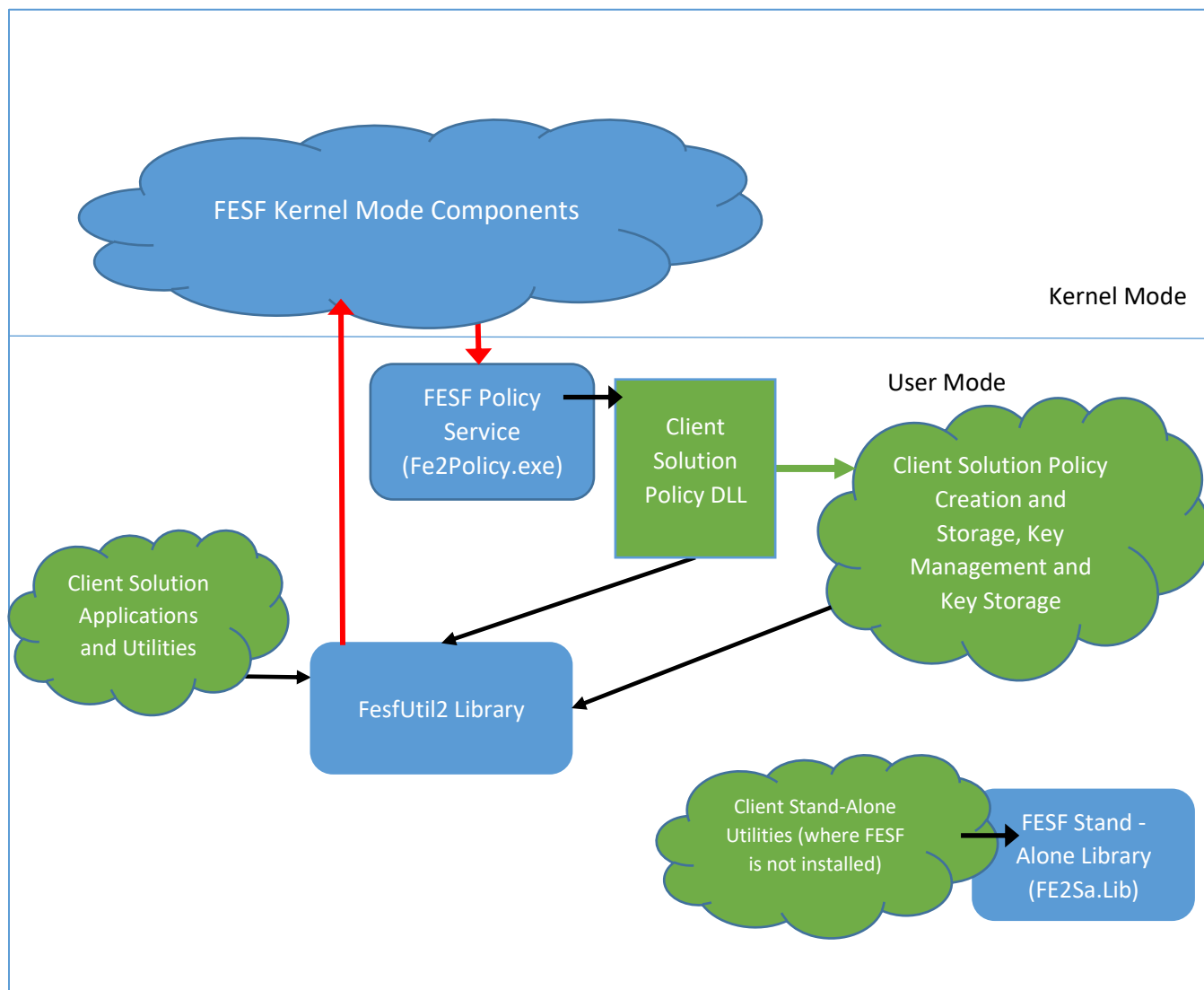
In terms of FESF Policy Caching, caching is done on a per-stream basis. Thus, on file systems that support alternate data streams, the support function **FesfUtil2PurgePolicyCache**, when called for a file, applies to a specific stream of the file (if the file has multiple data streams).

To repeat: 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.



## 5 FESF Components and Interfaces

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



**Figure 1 -- FESF Architectural 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 lines indicate FESF architecturally defined interfaces that are documented and supported by OSR. They represent a (synchronous) C/C++ function call and return interface.

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.



## 5.1 FESF Kernel Mode Components

The "big blue cloud" in Figure 1 represents the collection of OSR-supplied FESF Kernel Mode Components. We often refer to these as the "canonical FESF components." They comprise a set of file system Minifilters and their associated libraries. There are two kernel-mode components that are installed as part of FESF V2: FESFV2.sys and Fe2Lower.sys.

The FESF Kernel Mode Components are responsible for intercepting file operations (such as **CreateFile**, **ReadFile**, and **WriteFile**) on supported file systems and volumes, 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 FESF Standard License, but is available to licensees as part of the FESF Source Kit.

## 5.2 FESF User Mode Components

The orange rectangles in Figure 1 represent FESF User Mode Components. These components comprise the Fe2Policy service, the FesfUtil2 utility library, and the FESF Stand-Alone library (Fe2Sa.lib). OSR reserves the right to extend, add to, revise, or remove these components in future FESF releases

### 5.2.1 FESF Policy Service (Fe2Policy.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 Solution Policy DLL. All calls from the FESF Policy Service into the Solution 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 Fe2Policy, which in turn calls the Solution Policy DLL. While Solution components can call FESF to request information or perform utility functions, all calls that originate from FESF come through Fe2Policy and thereby to the Solution Policy DLL.

The FESF Policy Service calls entry points in the Solution Policy DLL in the context of a worker thread. These calls are synchronous. That is, the Solution Policy DLL must only return when it has the information requested (or else return an error). When a call to the Solution Policy DLL is made, that call is blocking an associated kernel-mode file operation. When the Solution 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 Fe2Policy from source, are provided as part of your FESF License. This code is for reference only. OSR does not support changes or customizations to Fe2Policy.

The specifics of the interface between Fe2Policy and the Solution Policy DLL, are described later in this document in the section [Solution Policy DLL Callback Function Reference](#).

### 5.2.2 FesfUtil2 Library

FesfUtil2 provides support functions to the Client Solution. The FesfUtil2 Library may be accessed by Client Solutions either as a statically linked library or as a shared DLL. See the documentation on [FesfUtil2](#) later in this document.

The FesfUtil2 Library provides general utility support to Client Solutions, as its name implies. A few of the services that FesfUtil2 provides are:

- 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 Fe2Policy, to a fully qualified local path specification.

- Retrieving the fully qualified path of a running application, given a Thread ID provided by Fe2Policy.
- Reading or updating the Solution Header Data that is stored on an FESF encrypted file.
- Globally enabling or disabling FESF Policy Caching.

The source code, and all the items necessary for building the FesfUtil2 Library from source are provided for your reference as part of your FESF License. This code is provided for your reference only. OSR does not support changes or customizations to FesfUtil2.

Additional details about FesfUtil2, including documentation for all the functions it makes available, is provided in the section entitled [FesfUtil2 Function Reference](#) later in this document.

### 5.2.3 FESF Stand-Alone Library (Fe2sa.lib)

The FESF Stand-Alone library provides support for implementing Client Solution components *on systems where FESF is not running*. Fe2Sa.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 may be useful when FESF is not installed, such as during a recovery operation or when dealing with FESF encrypted files on a Linux client that does not have FESF for Linux installed. 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 Fe2Sa from source, are provided as part of your FESF License. This code is for reference only. OSR does not support changes or customizations to Fe2Sa. Also, OSR does not support the direct use of functions that are internal to Fe2Sa.

***It is critically important to note that the FESF Stand-Alone library functions are not supported, and absolutely cannot be safely used, on systems where FESF is running. Even though calls to Fe2Sa functions on systems where FESF is running may appear to work properly, using the Stand-Alone library on systems where FESF is running can lead to file corruption.***

Any time you use the Fe2Sa functions on a system where FESF is installed but not running (for example, during installation to bulk encrypt a group of files immediately after FESF has been installed but before the system has been rebooted), it is critical that the system be rebooted before FESF is started to avoid any potential cache coherency issues.

Additional details about Fe2Sa.lib, including documentation for all the functions it makes available, are provided in the section entitled FESF Stand-Alone Library Function Reference later in this document.

## 5.3 Client Solution Components

The Client Solution will comprise as few or as many components as required to implement 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 Solution Policy DLL that will be called by the FESF Policy service.

### 5.3.1 Client Solution Policy DLL

The Client Solution Policy DLL is provided by the Client. OSR includes a complete and well-documented sample Solution 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 Solution 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 Solution Policy DLL are called by pointer. The Solution Policy DLL passes pointers to each of its callback functions during initialization processing. After initialization, FESF calls callback functions in the Solution Policy DLL to determine policy for a particular open instance of a file, as well as to retrieve the Solution Policy DLL defined Solution Header Data and Key data for files that are to be encrypted/decrypted by FESF.

As an example of how things work, consider the Solution 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 Solution 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 Solution Policy DLL's *PolGetKeyNewFile* to retrieve the Algorithm ID, Key, and Solution Header Data. During the call into the Solution 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 Solution Policy DLL must be prompt.

Processing for other callbacks in the Solution 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 application is informed of the result. Again, this call into the Solution Policy DLL is blocking completion of Windows' kernel mode processing of this open operation and ultimately the application's further progress.

#### 5.3.1.1 Note on lack of serialization among Solution 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 Solution Policy DLL can take place in parallel. FESF provides no serialization for calls into the Solution Policy DLL. Thus, it will be typical for multiple threads to call into the Solution Policy DLL simultaneously. In fact, it is even possible for the Solution Policy DLL to get multiple simultaneous calls to the same callback function, such as *PolGetPolicyNewFile* for the same file. This is possible when multiple threads attempt to access a file at the same time.

In the case that two different threads both simultaneously attempt to create the same (new) file, only one of them will ultimately succeed (in kernel-mode processing). FESF will ignore the result returned by the Solution Policy DLL from the second (and, hence, unsuccessful) attempt. Subsequent attempts to retrieve the Key Data and Solution Header Data will consistently return the same Key Data and Solution 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 Solution 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 Solution Policy DLL's *PolGetPolicyExistingFile* callback will all be relevant.

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

## 6 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 Solution Policy DLL. In this section, we'll describe the interface functions and major points to consider in implementing a Solution Policy DLL.

The FESF Policy Service calls callback functions in the Solution Policy DLL to determine Policy, gather data, provide an opportunity for the Solution Policy DLL to exercise control over particular functions, and to perform other support operations. The Solution 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 **Fe2Policy**. After the Solution Policy DLL is loaded, FESF calls **PolicyDllInit** to allow the Solution Policy DLL to perform initialization processing. This initialization includes the Solution Policy DLL calling **FePolSetConfiguration** to inform FESF of various configuration choices, including providing pointers to FESF for the callback functions that the Solution Policy DLL supports.

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

In other Solution architectures, Clients may design their Solution Policy DLL to be a more active participant in policy determination. In these architectures, the Solution 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 Solution Policy DLL has its particular advantages and disadvantages. The approach 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 Solution 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**.

### 6.1 Solution Policy DLL Callback Functions

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

- **PolicyDllInit** – This is the only entry point into the Solution Policy DLL that FESF calls by name, and it must be named **PolicyDllInit**. This callback function is called by FESF after the Solution Policy DLL has been loaded to allow the Solution Policy DLL to perform initialization processing. This processing must include initializing a **FE2\_POLICY\_CONFIG** structure and filling it in with pointers to the other callback functions supported by the Solution Policy DLL. The **FE2\_POLICY\_CONFIG** must then be passed to FESF by the Solution Policy DLL calling **FePolSetConfiguration** during its **PolicyDllInit** callback function processing. This callback function is required and must be implemented by every Solution Policy DLL.
- **PolAttachVolume** – Called when a new volume or network share is discovered by FESF (for example, when a thumb drive is inserted, a disk is formatted, or a network share is mapped). This gives the Solution Policy DLL the

option to have FESF support encryption/decryption on the volume or to have FESF ignore future file operations on the volume. This callback function is optional.

- *PolGetPolicyNewFile* – Called when a new file is being created on a volume to which FESF has attached, to determine if the file should be created in FESF encrypted format. This callback function is required and must be implemented by every Solution Policy DLL. Note that for the purposes of FESF, a "new file being created" includes 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, as well as the Solution Header Data that will be stored with the file. This callback function is required and must be implemented by every Solution 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 Solution Policy DLL.
- *PolGetKeyFromHeader* – Called when an existing FESF encrypted file is being opened, and the Solution Policy DLL has previously indicated (in a prior call to *PolGetPolicyExistingFile*) that the opening handle will receive transparent encrypted/decrypted access. Given the Thread ID of the thread performing the access and the Solution Header Data that FESF stored with the file, the Solution 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 Solution Policy DLL.
- *PolFreeHeader* – Called by FESF to return the storage for Solution Header Data that was previously allocated by the Solution Policy DLL. This callback function is required and must be implemented by every Solution Policy DLL.
- *PolFreeKey* – Called by FESF to return the storage for the Key Data that was previously allocated by the Solution Policy DLL. This callback function is required and must be implemented by every Solution Policy DLL.
- *PolApproveRename* – Called when a file on a supported file system is being renamed. The Solution 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 Solution Policy DLL can choose to allow or disallow the operation for security purposes. This callback function is optional.
- *PolApproveTransactedOpen* – Called when a new file is being created using **CreateFileTransacted** or similar. The Solution Policy DLL can choose to allow or disallow the operation for security purposes. This callback is optional.
- *PolGetPolicyDirectoryListing* – Called when a directory is opened to determine whether the sizes returned in a directory listing will reflect what is consumed on disk (including space used for the Solution Header Data and any metadata that FESF itself stores) or just the size of the data in the file (reflecting the size of the file if it were not FESF encrypted). This callback function is optional.
- *PolReportLastHandleClosed* – Called when the last handle to a file in FESF encrypted format is being closed. This callback function is optional.
- *PolGetLockRounding* – Called to find the "lock rounding" that should be associated with all accessors to a (network hosted) file prior to the locking request being sent to the remote server. This callback function is optional.
- *PolUninit* – Called during shutdown to allow the Solution Policy DLL to perform any cleanup operations it requires. This callback function is optional.

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

## 6.2 FESF Policy Before the Solution Policy DLL Starts

It will be no surprise that during system startup, Windows opens, creates, and renames numerous files. What does FESF do, in terms of Policy, before the Solution Policy DLL has been started and initialized?

Before the Solution Policy DLL starts, FESF implements secure defaults that will still allow the system to start and get work done. Access to all volumes and network shares will be monitored by FESF (until the Solution Policy DLL's *PolAttachVolume* callback can be called). During this time, new files will be created raw and any attempt to access an existing FESF encrypted file will be denied. In addition, rename and hard link operations will be allowed and directory enumerations will return raw file sizes (that is, the size including the FESF metadata and Solution Header Data)

## 6.3 Solution Policy DLL Initialization

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

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

To be able to call *FePolSetPolicyConfiguration*, the Solution Policy DLL builds an **FE2\_POLICY\_CONFIG** structure. This structure is typically allocated on the stack by the Solution Policy DLL. The structure must be zeroed before use.

Also specified in the **FE2\_POLICY\_CONFIG** structure are a list of encryption algorithms and options, and a unique handle that will be used by the Solution Policy DLL to identify each specific algorithm/option pair.

## 6.4 Returning Failure from Solution Policy DLL Callbacks

The Solution Policy DLL callbacks *PolGetPolicyNewFile*, *PolGetKeyNewFile*, *PolGetPolicyExistingFile*, and *PolGetKeyFromHeader* can all return failure indications. Solutions should avoid returning failures to these callbacks as a means of access control. Rather, returning failure from these functions should be reserved only for significant 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 Solution 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 Solution Policy DLL's callback is called.

In these cases, returning an error from one of the previously 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.

## 6.5 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 Solution Policy DLL callbacks. Those four things, in no particular order, are:

- **All Solution Policy DLL callback code that you implement must be thread-safe.** Fe2Policy uses Windows thread pools which dynamically grow (and shrink) the pool of worker threads it uses to call Solution Policy DLL callbacks. 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 Solution Policy DLL must complete "promptly."** Remember, when Fe2Policy calls your Solution 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 own 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 Solution 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 Solution Policy DLL callback is rarely desirable, as further described below.

- **Returning failure codes only when unrecoverable errors occur.** As described in the section above entitled Returning Failure from Solution Policy DLL Callbacks, returning a failure code from the *PolGetPolicyNewFile*, *PolGetKeyNewFile*, *PolGetPolicyExistingFile*, and *PolGetKeyFromHeader* Solution Policy DLL callbacks can have unexpected side-effects. 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 consider returning an error from these callbacks as an alternative way of implementing file access security.
- **Be conscious of the potential for reentrancy from your Solution and avoid it.** Bear in mind that your Solution 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 create a new file as a result of being called in your *PolGetPolicyNewFile* callback. In this example, assuming your policy determinations are made in a separate process (similar to the way the OSR Sample Solution works) your *PolGetPolicyNewFile* callback would get called-back endlessly (each time, creating a new file which then results in another callback). Not good!

To preserve FESF integrity (and to eliminate the most obvious potential causes of endless callbacks), FESF kernel-mode code suppresses all Solution Policy DLL calls for I/O operations that are performed within the context of the FESF Policy Service itself. So, in the above example, creating a new file directly within the Solution Policy DLL's *PolGetPolicyNewFile* callback (and not in a separate process) would NOT actually generate a reentrant call to *PolGetPolicyNewFile*.



## 6.6 Guidance Regarding Solution Policy DLL Solution Header Data

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

The Solution Policy DLL defined Solution Header Data may contain any data the Solution may require to derive the encryption key information for the file on subsequent accesses.

While Solution Header Data can be any size up to **FE\_MAX\_SOLUTION\_HEADER\_SIZE** (which is currently 1MB), the Solution Policy DLL must correctly fill-in the **MaxHeaderSize** value in the **FE2\_POLICY\_CONFIG**. The Solution Policy DLL must set this field to (at least) the largest Solution Header size in bytes that the Solution will exchange, in either direction, with Fe2Policy. This parameter is used by FESF to size pre-allocated buffers for communication between components.

Oversizing the **MaxHeaderSize** field will result in wasting pre-allocated memory on machines where FESF is installed. If FESF V2 encounters a Solution Header that is larger than the value supplied by **MaxHeaderSize**, runtime errors and unpredictable application behavior will occur.

It is important to understand that **MaxHeaderSize** is a *run time parameter* used for communication by FESF. It is not stored in FESF encrypted files nor is it used to size on disk structures. This means that a given version of a Solution can set **MaxHeaderSize** to one value, but if a later version of the Solution uses a larger header size **MaxHeaderSize** can be changed without any complications or compatibility issues with previous versions.

Finally, please note that the only supported method for a Solution to retrieve or update the Solution Header Data stored with an FESF encrypted file is by using documented functions, such as those supplied by FesfUtil2 (while FESF is installed) or Fe2Sa (when FESF is not running on the system) and described later in this document.

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

File paths and other associated open information are expressed to the Solution 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 Solution 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 Solution Policy DLL may also call the FESF Utility function **FesfUtil2GetFullyQualifiedPath**, 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 Solution Policy DLL interface and as input to the FesfUtil2 function **FesfUtil2GetFullyQualifiedPath**. 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 named 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\Doe.docx`
- `\\EmployeeFiles.FESFTest.com\Records\Doe.docx`
- `\\10.0.0.10\Records\Doe.docx`
- `\\10.0.0.11\Records\Doe.docx`

In each of these cases, the Solution Policy DLL will see the server name as specified by the requestor. Thus, if the Solution is using a strict path-based approach to determining policy, these paths may appear to represent different files located on different servers. If the Solution requires a unique canonical name to represent the server, the Solution 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 their Z: drive to the `\\EmployeeFiles\Records` share. If the user then accesses `Z:\Doe.docx`, the Solution Policy DLL will not see the drive letter based path in its callbacks. As stated previously, FESF always passes the Solution 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 may at times get unexpected results, including result values from a function such as **FesfUtil2GetExecutablePathForThreadId**.

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 Solution Policy DLL interface and as input to the **FesfUtil2** function **FesfUtil2GetFullyQualifiedPath**. 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
```

## 6.8 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 Solution Policy DLL supplied Solution Header Data is via documented FESF functions (such as **FesfUtil2ReadHeaderExclusive** and **FesfUtil2UpdateHeaderExclusive**).

It is a serious architectural violation for a Solution Component to bypass the supported FESF functions and instead use raw access to manually update (or attempt to interpret) any portion of the FESF ODS, including the Solution Header Data. Thus, if you write code that accesses, reads, interprets or changes the stored FESF metadata or Solution Policy DLL supplied Solution Header Data in any way other than using the documented FESF functions, you're doing this against our recommendations, and you are on your own. Consider that FESF may have header data for a given file cached in kernel-mode at any time. Also, note that OSR reserves the right to change the FESF On Disk Structure at any time, for any reason or for no reason, with or without notice.

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.

## 6.9 Installing and Using Fe2Policy

### 6.9.1 Installation, Removal, and Specifying Your Solution Policy DLL location

The Fe2Policy service is self-installing. This is the only supported method of installing the Fe2Policy service. To install Fe2Policy, use the command:

```
Fe2Policy --install [logging options] <fully qualified path to your Solution Policy DLL>
```

Note that the "install" option must be preceded by two dashes.

As part of installing Fe2Policy, you specify the fully qualified path to your Solution Policy DLL. Your Solution Policy DLL forms a critical part of the FESF security infrastructure. Be sure you locate it securely in a directory that is protected from unauthorized access. It would be a security vulnerability if a bad actor created an alternate Solution Policy DLL with the same name as yours and got Fe2Policy to load it.

You may optionally add logging options to Fe2Policy's install command line to specify the Windows Event Log logging level and/or on disk log location and level, as described later in this document.

Fe2Policy installs itself to start at "auto-start" time. Fe2Policy is also set to automatically re-start itself if it exits for any reason other than an ordered stop or shutdown. Installing Fe2Policy does not start it. The system must be rebooted to properly start Fe2Policy.

Fe2Policy can be un-installed using the standard Windows SC command, such as:

```
sc.exe delete Fe2Policy
```

This un-installs the Fe2Policy service, deleting its entries from the Registry, but does not delete the Fe2Policy executable associated with the service.

### 6.9.2 Fe2Policy Logging Options

Fe2Policy logs messages to the Windows Event Log. These messages can be seen in the Windows Event Viewer under "Windows Logs", "Applications" (see Figure 1). By default, Fe2Policy will log messages related to its state (startup, shutdown, etc.) and any errors that it encounters.

The level (verbosity) of the messages that are written to the event log can be specified using the "-e=<level>" Fe2Policy option, where <level> is "i", "w", or "e" to log "informational", "warning", and "error" events respectively. Lower levels include higher levels, thus specifying "-e=i" will result in information, warning, and error events from Fe2Policy to be written to the Windows Event Log.

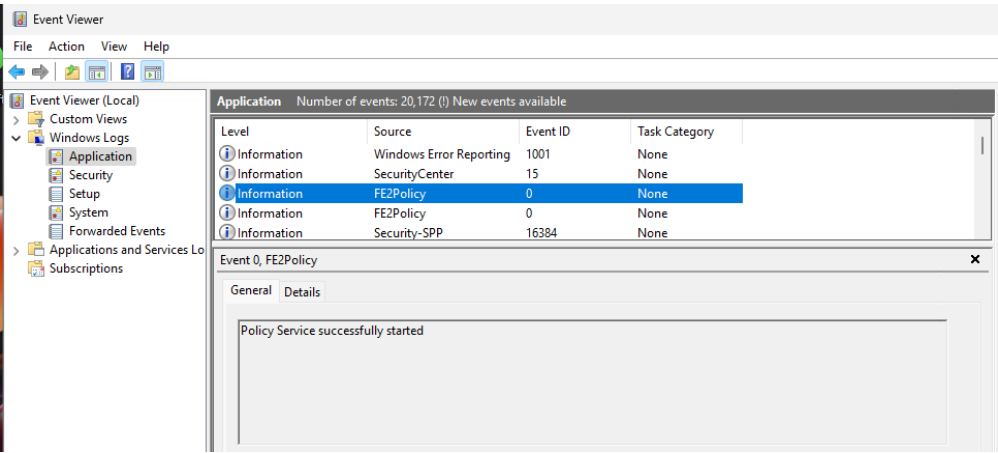


Figure 2 - Windows Event Log for Fe2Policy

In addition to writing events to the event log, Fe2Policy can optionally log information to its own on-disk log. This allows you to collect detailed log information for debugging or diagnostic purposes, without cluttering the Windows event log.

To specify on-disk logging, you use the Fe2Policy options “-f=<log file path> -l=<level>”, where <log file path> is the fully qualified path where Fe2Policy should write its log, and <level> indicates the level (verbosity) of the messages written to the on-disk log. Note that Fe2Policy automatically rotates logs when they reach about 512MB in size. This size is not configurable. An example of the Fe2Policy log output is shown in Figure 3.

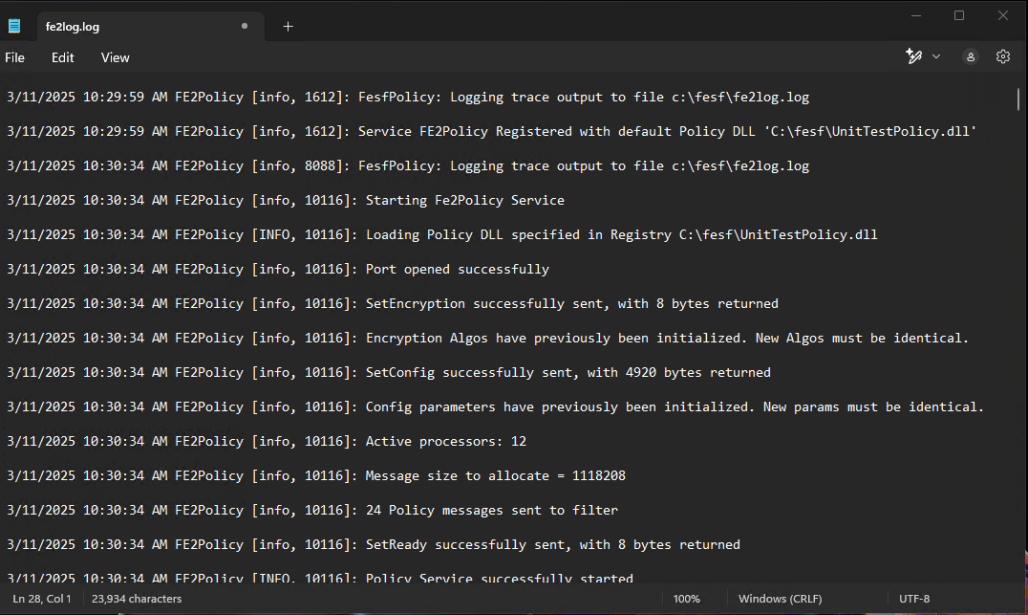


Figure 3 - Fe2Policy Informational Level Log Output

Note that the event messages that are written by Fe2Policy to the Windows Event Log and its on-disk log are the same. So, for example, if you choose to log “error” level messages to the Windows Event Log (with the option -e=e) and “warning” level messages to the Fe2Policy private on-disk log (using the -l=w option), the Fe2Policy on disk log will contain all the “error” level messages that Fe2Policy has written to the Windows Event Log as well as any

“warning” level messages (that are not written to the Windows Event Log). Finally, be aware that the logging data that is output at the informational level is quite granular, and much of it might only be of use to OSR or to source code licensees.

### Fe2Policy Option Summary

Option Short Form	Option Long Form	Meaning
-l=<t i w e>	--logging=<t i w e>	File logging output level, one of: t(race), i(nfo), w(arning), e(rror)
-f=<log-file-path>	--file=<log-file-path>	Logs file logging messages to specified file. If no level specified defaults to “-l=e”
-e=<i w e>	--eventlog=<i w e>	Windows event log output level, one of: i(nfo), w(arning), e(rror). If not specified, defaults to “-e=e”
(none)	--install	Installs the service (MUST specify path to default Solution Policy DLL)

#### 6.9.3 Updating Fe2Policy Solution Policy DLL Path or Logging Options

After installing Fe2Policy, there are two ways to update the Logging Options or the path to your Solution Policy DLL:

1. Uninstall the Fe2Policy Service (“sc.exe delete Fe2Policy”) and then re-install Fe2Policy again using the self-install feature described previously. This is the only supported method, because it validates the logging options and Solution Policy DLL location and makes the entries in the Windows Registry in the proper format.
2. Directly edit the Fe2Policy Registry entries. For your reference, we describe the Registry entries used by Fe2Policy later in this document. However, be aware that getting the syntax of those entries correct (especially when there are spaces in the path to Fe2Policy or your Solution Policy DLL) can be tricky. If you must edit the Registry entries, be careful when doing so, and bear in mind that this is not supported by OSR.

We recommend that if you need to update the Solution Policy DLL, its location, or the Fe2Policy logging options that you stop Fe2Policy, uninstall it using the Windows SC command, make your changes using Fe2Policy’s self-install feature, *and then reboot the system for the new changes to take effect.*

While you can simply restart the Fe2Policy service (“sc start Fe2Policy”) you’ll need to be aware of three things:

- You must stop and restart the FESF Policy Service (Fe2Policy.exe) for the new Solution Policy DLL to be loaded or any changes to the selected options to be recognized. The FESF Policy Service only loads the Solution Policy DLL and evaluates options during initialization.
- Be aware that although you’ve restarted Fe2Policy, the FESF kernel-mode components (and Windows) will have “state” that is maintained and is not reset as a result of the FESF Policy Service being restarted. This means, for example, that volumes that been previously presented to your Solution Policy DLL via the *PolAttachVolume* callback will not be presented to your Solution Policy DLL again. Similarly, any files that were encrypted by FESF by previous instances of the Solution Policy DLL will be recognized as encrypted files by FESF. Your Solution Policy DLL may, therefore, be called at *PolGetKeyFromHeader* with Solution Header Data that was created by a previous Solution Policy DLL. It is therefore wise to include some identifying information in the Solution Header Data so that your Solution Policy DLL can validate and recognize the

header before using its contents.

- The configuration parameters provided by your Solution Policy DLL via the FE2\_POLICY\_CONFIG structure must be identical each time you start Fe2Policy, unless the system has been rebooted. For example, you cannot stop Fe2Policy, update your Solution Policy DLL to support an additional crypto algorithm, and then restart Fe2Policy. In this case, Fe2Policy will fail with **STATUS\_DLL\_INIT\_FAILED** and an error reflecting this will be logged to the Windows Event Log. To continue the example, to add support for an additional crypto algorithm you would stop Fe2Policy, change your Solution Policy DLL to add the new crypto support, and then reboot the system.

#### 6.9.4 Fe2Policy Registry Entries

While we strongly recommend that you change Fe2Policy options using Fe2Policy's self-install feature as described previously, for your reference the FESF Policy Service uses the Registry values described in this section.

##### 6.9.4.1 Fe2Policy Options

Policy options used by the FESF Policy Service can be specified in the ImagePath value in the Fe2Policy Service entry:

Key: HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Fe2Policy  
 Value Name: ImagePath  
 Value Type: REG\_SZ  
 Value: <Fully qualified path to Fe2Policy><any options>

For example:

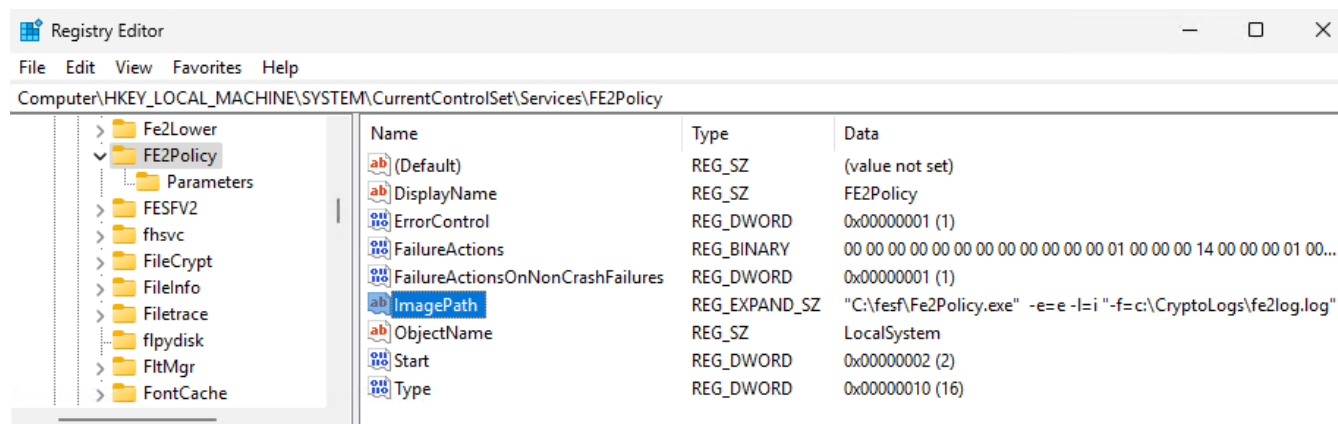


Figure 4 -- Fe2Policy ImagePath with Options

In the above screen capture, the ImagePath value in the registry specified the fully qualified path to the Fe2Policy executable (C:\fesf\Fe2Policy.exe) and requests that error messages be logged to the Windows Event Log (-e=e) and that informational, warning, and error messages be written to a file (-l=i, -f=c:\CryptoLogs\fe2log.log).

##### 6.9.4.2 Solution Policy DLL

Fe2Policy stores the fully qualified path to your Solution Policy DLL in the following registry value:

Key: HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Fe2Policy\Parameters  
 Value Name: PolicyDll  
 Value Type: REG\_SZ  
 Value: <Fully qualified path to the Solution Policy DLL>

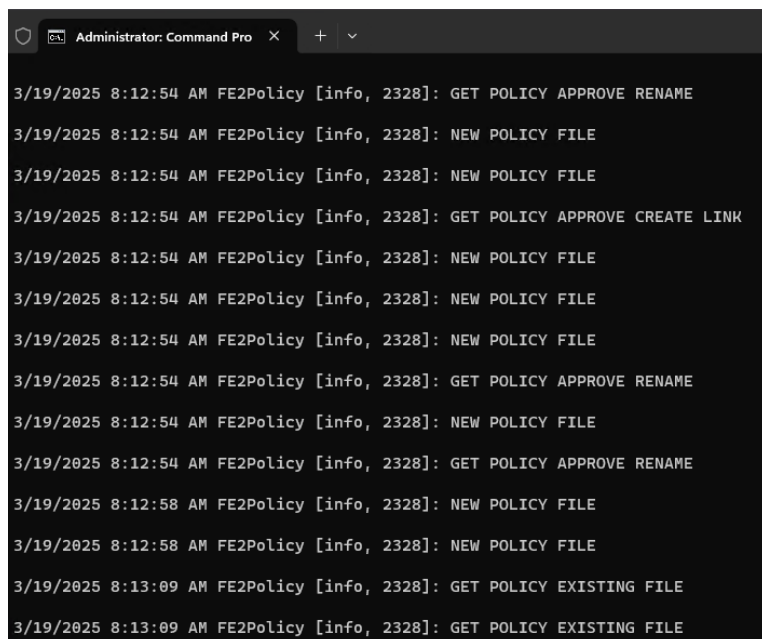
To avoid potential security vulnerabilities, it is important to always specify the fully qualified path to your Solution Policy DLL, and to locate your Solution Policy DLL in a directory that is protected by Windows Access Control Lists from unauthorized access.

### 6.9.5 Using Fe2Policy for Debugging

For the purpose of debugging your Solution (particularly, your Solution Policy DLL), Fe2Policy can be run from the command line instead of as a service. You can accomplish this by stopping the Fe2Policy service and starting Fe2Policy from the command line with no arguments. Alternatively, you can start Fe2Policy from the command line and specify the fully qualified path to an alternate Solution Policy DLL.

When you run Fe2Policy from the command line, it displays “information” level and higher logging output in the terminal window. Every call to the Solution Policy DLL is logged (except for calls to *PolGetPolicyDirectoryListing*, which are too numerous to log usefully). This terminal windows output is produced in addition to any logging that you specify should be sent to the Windows Event Log or a Fe2TraceLog.

An example of Fe2Policy logging output is shown in Figure 5.

A screenshot of a Windows Command Prompt window titled "Administrator: Command Prom...". The window displays a series of log messages from Fe2Policy. The messages are timestamped and include the process name, log level, and a message ID. The log entries show various operations such as "GET POLICY APPROVE RENAME", "NEW POLICY FILE", and "GET POLICY EXISTING FILE".

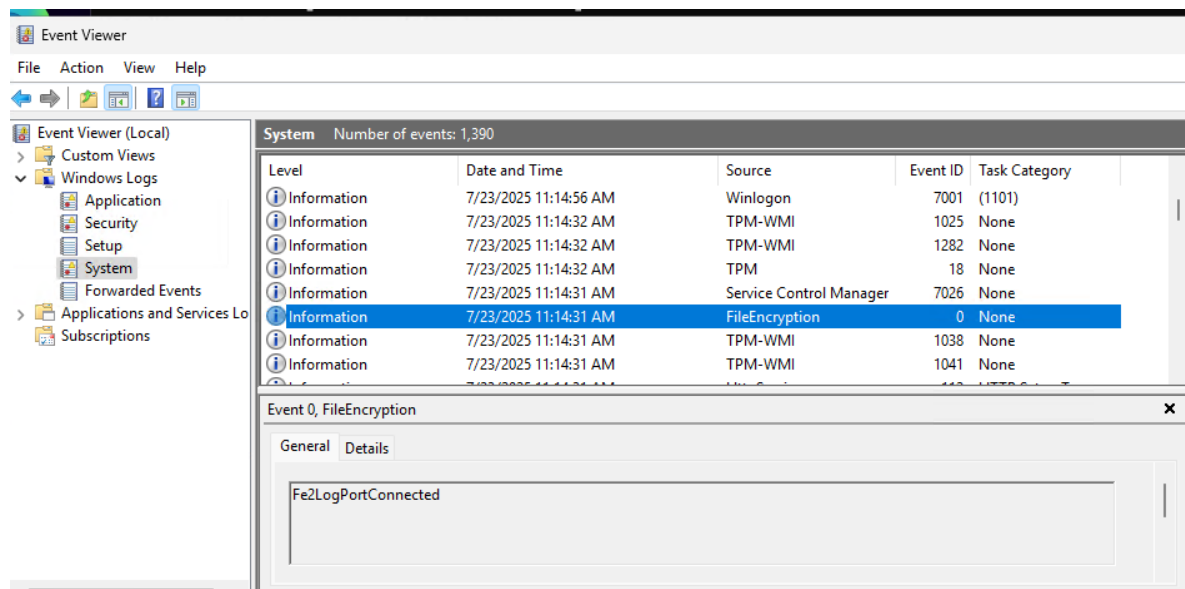
```
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: GET POLICY APPROVE RENAME
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: GET POLICY APPROVE CREATE LINK
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: GET POLICY APPROVE RENAME
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:12:54 AM FE2Policy [info, 2328]: GET POLICY APPROVE RENAME
3/19/2025 8:12:58 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:12:58 AM FE2Policy [info, 2328]: NEW POLICY FILE
3/19/2025 8:13:09 AM FE2Policy [info, 2328]: GET POLICY EXISTING FILE
3/19/2025 8:13:09 AM FE2Policy [info, 2328]: GET POLICY EXISTING FILE
```

Figure 5 -- Running Fe2Policy From the Command Line

## 6.10 FESF Kernel Component Logging and Tracing

FESF V2 logs various events to the Windows Event Log. While FESF generally avoids logging routine information to the Event Log, it does log each time Fe2Policy connects to the kernel-mode FESF driver. This allows you (and your users) to know when FESF is successfully started and ready for work. FESF also logs any serious errors to the Windows Event Log.

Unless you have an FESF Source License, the name of the logging provider is not customizable. Because licensees may customize the names of the FESF components when they integrated them into their Solutions, FESF uses the name “FileEncryption” as the source of its messages in the kernel event log.



**Figure 6—Log entry from FESF Kernel Mode Components. Note the source name “FileEncryption” (not FESF)**

FESF V2 includes a low overhead event tracing capability based on Event Tracing for Windows (ETW). Tracing facilities are present in both the debug and release builds of the FESF kernel-mode components but are not active until they are enabled. This tracing capability is designed exclusively for OSR’s use in diagnosing problems in the field. The low overhead nature of the tracing capability allows it to be enabled while reproducing a problem, even on end-user systems, with little noticeable performance impact.

When necessary, OSR will request that you enable tracing, reproduce the problem, stop tracing, and then send the collected trace file to OSR. Trace files are stored in a binary format to make them space efficient. Please do not collect a trace file unless specifically requested by OSR.

Trace sessions can be controlled using the logman utility which is available by default on all Windows systems.

To create and start a trace session, open an administrator command prompt windows and use a command such as the following (or an alternative, if specifically requested by the OSR):

```
logman create trace FE2 -p {E8887935-6D20-4A4E-B287-B28D3D8F4F7B} 1 255 -bs 64 -ets
```

After the above command has been successfully run, reproduce the problem. Problems should always be reproduced on systems that are as idle as possible (in terms of file system activity) while the problem is being reproduced. This is critical to allowing the OSR engineering team to identify the specific issue.

After the problem has been reproduced, stop the trace session with a command similar to the following:

```
logman stop FE2 -ets
```

The above commands will create a file named FE2.etl (located in the directory from which logman was run to create the trace). Note that the trace output file is a binary file. Please send this file to OSR to aid in diagnosis.

## 7 Building FESF From Source

OSR supports building FESF V2 using Visual Studio 2022 V17.14. OSR does not support building any of the FESF components using any other tools or versions of Visual Studio, including Visual Studio 2019.

The layout of the FESF V2 source directory is as follows:

\src	The top level source code directory for FESF; Contains no files
\customize	Contains FE2_Customize.h (for customizing Fe2Policy)
\inc	Headers shared between FESF user and kernel mode components
\kernel	FESF canonical kernel mode components
\shared	Shared source code between FESF user and kernel mode components
\user	FESF canonical and sample user mode components
\Fe2User	The FESF canonical user-mode components
\Unsupported_Sample	The source code for the FESF Sample Solution
\inc.user	User mode common include files
\lib.user	User mode common library files

### 7.1 FESF Canonical User-Mode Components

The source code for the FESF V2 canonical user-mode components (the ones shown as orange rectangles in Figure 1) can be found in the directories rooted at

```
\src\user\Fe2User
```

You will find source code and projects for building Fe2Policy and the FesfUtil2 libraries (both static and dynamic). You must build all of these components successfully before attempting to build any of the FESF Sample Solution components.

Sources in these directories may reference and/or produce output files in the user-mode common directories:

```
\src\user\inc.user
```

```
\src\user\lib.user
```

Sources in these directories may also reference files in the FESF shared kernel/user directory:

```
\src\inc
```

The user-mode components can be built using the single Visual Studio Solution file:

```
\src\user\Fe2User\Fe2User.sln
```

The sources build without warnings or errors and pass Code Analysis as supplied.

### 7.2 FESF Sample Solution Components

Note that the Sample Solution Components are dependent on the FESF canonical user-mode components. Therefore, you must successfully build the FESF canonical user-mode components before you can successfully build the FESF Sample Solution components.



The Fe2SaCrypt sample cannot be successfully built without binaries and headers for the **SymCrypt** library. **SymCrypt** is available on GitHub as both [source code](#) and [pre-built binaries](#).

The sample installer, as provided, cannot be successfully built without binaries for the FESF Kernel Mode components. You can copy the released version of the required components to the required directories (see the sample installer) before building the sample installer.

The source code for the FESF V2 Sample Solution can be found in the directories rooted at

```
\src\user\Unsupported_Sample
```

You will find source code and projects for building all the components of the FESF Sample Solution. Also, note that OSR supplies an installer that will install the FESF canonical components and the sample solution. That installer is located in:

```
\src\user\Unsupported_Sample\Installer
```

You may use this installer as a guide for creating an installer for your product, as it demonstrates the proper (and only supported) method for installing FESF kernel-mode and user-mode components.

The Sample Solution sources may reference files in the Sample Sources shared directory:

```
\src\user\Unsupported_Sample\inc
```

The Sample Solution source code may also reference files in the user-mode common directories:

```
\src\user\inc.user
```

```
\src\users\lib.user
```

They may also reference files in the FESF shared kernel/user directory:

```
\src\inc
```

The Sample Solution components can be built using the single Visual Studio Solution file:

```
\src\user\Unsupported_Sample\Sample.sln
```

This Solution builds the Sample Solution components including the FesfSaCrypt utility and the Sample Installer (neither of which will build successfully as provided, without manually adding the required components described above).

The sources build without warnings or errors and pass Code Analysis as supplied.

### 7.3 FESF Kernel Mode Components

FESF kernel mode components can be built by licensees with FESF V2 source code licenses using the FESFV2 Solution located in the `src\kernel\` directory. Recall that, absent a custom agreement, an FESF source license does not include support by OSR for modifying any FESF kernel mode components. Assistance in understanding the FESF kernel mode components, or in designing or implementing custom changes to the FESF kernel mode components, is available only with an additional, custom, engagement with OSR.



## 8 Notes on Installing FESF with Your Solution

Please see the `src\user\Unsupported_Sample\Installer` project (part of the Sample Solution) for an illustration on how to install the FESF Components and other components. This installation process uses OSR supported procedures.

Note that the two kernel-mode components, `FesfV2.sys` and `Fe2Lower.sys`, must be started at **System Start time**. This is a change from FESF V1. Also, please be aware that OSR does not support changing or installing the kernel-mode components without rebooting the system after the kernel-mode components have been installed. That is, it is not permissible to stop `Fe2Policy` and unload the kernel-mode components, and then reload the kernel-mode components and restart `Fe2Policy`. The system must be rebooted after the kernel-mode components have been stopped.

### 8.1 About the Sample Installer Project

As mentioned previously, OSR provides an unsupported sample project that builds an MSI file that installs both the FESF Canonical components and the Unsupported Sample Solution. We provide this project because over the years FESF licensees have frequently asked us for an example way to install various FESF components. The sample installation executable is built using [the WiX toolset](#). Our use of Wix is essentially arbitrary and is absolutely not required for the installation of the FESF Canonical components.

This project, along with the INF files provided with the drivers, does illustrate the correct, supported, way to install the FESF components. These are shown in the batch file “`InstallFesfV2.bat`” in the sample installer project. Specifically, the supported steps for installing FESF are:

- The FESF V2 drivers, `Fe2Lower.sys` and `FesfV2.sys` must be installed using their INF files. We recommend that you use the INFs to install the drivers via a procedure like the following (from “`installFesfV2.bat`”):

```
RUNDLL32.EXE SETUPAPI.DLL,InstallHinfSection DefaultInstall 128 .\Fe2Lower\Fe2Lower.inf
RUNDLL32.EXE SETUPAPI.DLL,InstallHinfSection DefaultInstall 128 .\FesfV2\FesfV2.inf
```

- You *must* set the “`enableecp`” property for the `LanmanServer` (Windows networking) component, to enable FESF to properly detect network volumes. Specifically:

```
REG_DWORD SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters\enableecp
```

Must be set to the REG\_DWORD value of “1”

- `Fe2Policy` must be installed using its self-install capability (“`Fe2Policy --install <your-policy-dll> <options>`”). For example (again from “`installFesfV2.bat`”):

```
Fe2Policy.exe --install -e=e Samppolicy.dll
```

- After the drivers or `Fe2Policy` has been updated, the system must be rebooted.

### 8.2 About Customizing FESF Component Names

Note that licensees may customize the names of any or all of the FESF components. The service name for `Fe2Policy` needs to be changed by specifying the new name in the file `src\Customize\FE2_Customize.h` and also the “Target Name” parameter in the `Fe2Policy` Project’s configuration. If you change the name of `Fe2Policy`, remember to rebuild your Solution Policy DLL after building the newly named `Fe2Policy`.

Drivers may be renamed in any manner that is convenient. Note that any changes you make to driver names or the INF file will render the OSR-applied signatures invalid.

## 9 FESF Known Restrictions and Limitations

---

For information related to the most recent versions of FESF, please refer to the Release Notes for:

- Currently supported versions of Windows
- Versions of Visual Studio and WDK that may currently be used by FESF

The restrictions and limitations below are permanent, documented, restrictions or limitations related to supported FESF V2 behavior.

- **32-Bit Windows Systems**  
FESF does not support any 32-bit Windows systems and does not provide or support a 32-bit version of the FESF Utility Library FesfUtil2.
- **Supported File Systems**  
FESF supports encrypting files on NTFS, REFS, FAT32, ExFAT (both locally and on the network). FESF does not support encrypting or decrypting FESF files on UDF or CDFS, any non-Microsoft file systems, or on network shares not hosted by SMB running a version of Windows that is currently covered by Microsoft's Modern Life Cycle for the General Availability Channel or Mainstream Support for the Long-Term Servicing Channel (for Enterprise LTSC/LTSB versions only).
- **Only Windows-Hosted SMB Shares Are Supported**  
Only network volumes shared from Windows systems (clients or servers) via SMB are supported. Non-SMB protocols such as HGFS (used to share between a VMWare client and its host) or RDPR (used to share between a Remote Desktop and the machine it's being accessed from) are explicitly ignored and no encryption will take place on them.
- **Concurrent Support for NTFS Compression or Encryption**  
FESF encryption does not support NTFS compression or encryption for FESF encrypted files. If an FESF file is properly encrypted, the data will be statistically random. The file, therefore, will not benefit from compression. Also, if the file is already encrypted by FESF, it will not benefit from further encryption.

Note that files that are NOT FESF encrypted can be successfully compressed or encrypted by NTFS. However, these files cannot subsequently be successfully encrypted by FESF without reversing the NTFS compression or encryption.

FESF not supporting files with NTFS encryption or compression is not an FESF defect but is rather by design.

- **Client-Side Caching (CSC)**  
Client-Side Caching (also known as Offline Files) is a legacy feature on Windows. While not officially deprecated by Microsoft at this time, it is in maintenance only mode and not recommended for use in favor of One Drive, Work Folders, or Cloud-Base Profiles.

While non-encrypted files may work with Client-Side Caching in FESF V2, FESF does not officially support Client-Side Caching (CSC).

- **Interactions with LUAFV**

In modern versions of Windows, the behavior of Microsoft's LUAFV filter has been significantly restricted. Even in supported scenarios, LUAFV redirection has always applied to only 32-bit applications that are not run as administrator. It is well known that ordinary operation of many antivirus filters will cause legacy 32-bit applications attempting to write to either of the "Program Files" directories or certain subdirectories of "Windows" to fail. In fact, even without any non-Microsoft antivirus filters installed, OSR's testing indicates that writes to protected directories fail by default in Windows 11.

We expect legacy 32-bit applications that attempt to create files in any of the virtualized directories to not be affected by FESF V2. As noted above, in recent versions of Windows 11 these operations fail with or without FESF installed. Regardless, just like typical antivirus filters, FESF does not officially support LUAFV redirection.

- **Interactions with Antivirus (and other file system filter) Products**

FESF has been successfully tested with many of the most common antivirus (AV), disaster recovery, and desktop virtualization products. Interoperability testing is an ongoing process for FESF. Nevertheless, licensees should expect that end users who have products that use file system filters will need to make these products aware of the both canonical FESF and Solution-specific components. We know this to be particularly true of antivirus products.

As an example of the configuration that should be expected, AV products should be configured to exclude the **FE2Policy** service (as well as the **FESFV2** and **FE2LOWER** Minifilters, if applicable) from any virus scanning or other processing. In addition, you may need to exclude any of your Solution's components of services from AV processing.

The requirement to configure the behavior of AV products does not reflect a bug or poor design of either the AV product or FESF. Rather, it is inherent in the way services and file system Minifilters interoperate. Excluding FESF components and your product's Solution components from AV processing should present minimal security threats to customer environments. Of course, every environment is different and requires unique analysis.

## 10 Solution Policy DLL Callback Function Reference

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

### 10.1 About Implementing Your Callback DLL

This section provides hints and tips for implementing your Solution Policy DLL callback functions.

#### 10.1.1 Regarding SAL Annotations

If you look at Fe2PolDllApi.h, where the FESF Solution 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_EX(
    _In_ FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_ DWORD ThreadId,
    _In_ PVOID context,
    _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/>

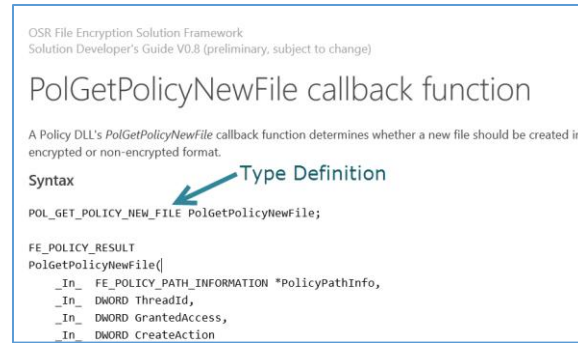
How do these annotations affect you? Well, mostly, they won't. Except that if you enable Visual Studio Code Analysis you'll be able to find problems with how you're implementing your Solution Policy DLL callbacks more quickly.

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 Solution 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 6 – Type Definition Example**

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

## Syntax

```
bool  
PolicyDllInit(  
    VOID  
)
```

## Parameters

(none)

## Return value

If **PolicyDllInit** succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

Returning **FALSE** will result in FESF not calling any further functions in the Solution Policy DLL.

## Remarks

All Solution Policy DLLs must implement this callback function. If this callback function is not implemented, the state and operation of FESF will be undefined. This callback must be named **PolicyDllInit** and is the only Solution Policy DLL callback that the FESF Policy Service locates by name.

A Solution Policy DLL's **PolicyDllInit** callback function is called by FESF to allow the Solution Policy DLL to perform initialization. The Solution 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 Solution Policy DLL within **PolicyDllInit** is to build an **FE2\_POLICY\_CONFIG** structure and pass a pointer to this structure to FESF by calling **FePolSetConfiguration**. The **FE2\_POLICY\_CONFIG** structure contains the following information:

- The version of the FE Policy Interface that the Solution Policy DLL supports.
- Whether FESF Policy Caching should be enabled by default (changeable at runtime).
- The maximum size (in bytes) of the Solution Header that the Solution Policy DLL supports.
- A list of one or more crypto algorithms, along with a unique string (the Algorithm ID) that will be used by the Solution Policy DLL to refer to each.
- Pointers to the Policy callback functions that are implemented by the Solution Policy DLL.

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

To enable clean-up operations FESF will call the Solution Policy DLL's *PolUnInit* callback function during shutdown.

Note that *PolUnInit* is called through the pointer provided in the **FE2\_POLICY\_CONFIG** structure and is not located by name.

## 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	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolApproveCreateLink callback function

A Solution Policy DLL's *PolApproveCreateLink* callback function is called to allow the Solution 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.

If FESF has determined that the source file is in a “bypass” region, then the **FE\_POLICY\_PATH\_NAME\_BYPASS** flag will be set. A bypass region is one in which file opens are always raw (because of interoperability issues).

If the requested rename is to replace an existing file (if one exists) then the **FE\_POLICY\_PATH\_REPLACE\_IF\_EXISTS** flag is set.

### *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 to 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.

If FESF has determined that the new name will be in a “bypass” region then the

**FE\_POLICY\_PATH\_NAME\_BYPASS** flag will be set. A bypass region is one in which file opens are always raw (because of interoperability issues).

## 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 Solution Policy DLL's *PolApproveCreateLink* callback function is called by FESF to allow the Solution 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 Solution Policy DLL. If this function is not implemented, FESF allows the proposed hard link to be created.

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 hard link 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.

## See Also

## Requirements

Software version	FESF V1 (or later)
Library	Fe2Policy.lib
Header	Fe2PoIDllApi.h

# PolApproveRename callback function

A Solution Policy DLL's *PolApproveRename* callback function is called to allow the Solution 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.

If FESF has determined that the source file is in a “bypass” region, then the **FE\_POLICY\_PATH\_NAME\_BYPASS** flag will be set. A bypass region is one in which file opens are always raw (because of interoperability issues).

If the requested rename is to replace an existing file (if one exists) then the **FE\_POLICY\_PATH\_REPLACE\_IF\_EXISTS** flag is set.

### *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 to 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.

If FESF has determined that the new name is “bypass”, then the **FE\_POLICY\_PATH\_NAME\_BYPASS** flag will be set. A “bypass region” is one in which file opens are always raw because of interoperability issues.

## 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 Solution Policy DLL's *PolApproveRename* callback function is called by FESF to allow the Solution Policy DLL to approve or reject a proposed rename operation for security reasons.

Implementation of this callback function is optional for a Solution Policy DLL. If this function is not implemented, FESF allows the rename.

Depending on how their Solution works, developers might choose to implement this callback (for example) to prevent files from being renamed into a directory where the files are all intended to be encrypted.

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.

## 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	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolApproveTransactedOpen callback function

A Policy DLL's *PolApproveTransactedOpen* callback function is called to allow the Policy DLL to approve or reject a transacted (or similar) open **for a new file**. All such opens are given RAW access by default. 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**).

*TransactionUnitOfWork* [in]

The ID (unit of work) of the transaction which this create is a part of.  
This will be all zeros if the FE\_POLICY\_PATH\_MULTIPLE\_VIEWS flag is set.

*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

The use of this callback has changed slightly starting in FESF V2. The new (V2.0 and later) behavior is described below.

Transaction support is a legacy feature that is supported only by NTFS and has been deprecated by Microsoft for at least ten years. FESF does not support transactional opens of FESF (new or existing) encrypted files. If an attempt is made to open an (existing) FESF encrypted file as part of an NTFS transaction, the request will be rejected with `STATUS_TRANSACTIONAL_CONFLICT`.

This callback is only called (a) for non-network files, and (b) for files created on volumes that support transactions. That means, practically speaking, that the Solution Policy DLL will only get this callback for files created on local NTFS volumes.

By default, when a new file is created within an NTFS transaction, the Policy DLL's `PolGetPolicyNewFile` callback is not called, and the file will be created raw. The Policy DLL's `PolApproveTransactedOpen` callback function allows the Policy DLL to approve or reject the transactional creation of a new, raw, file. A Policy DLL might do this (for example) to prevent the creation of a file in a directory in which all files are assumed to be FESF encrypted.

Implementation of this callback function is optional for a Policy DLL. If this function is not implemented, FESF will allow transactional create operations.

Solution developers should be VERY cautious about returning **FALSE** from this callback. Many critical parts of the system (notably Windows Update) still 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).

## See Also

## Requirements

Software version	FESF V1.1 (or later)
Library	Fe2Policy.lib
Header	PolDllApi.h



# PolAttachVolume callback function

A Solution Policy DLL's *PolAttachVolume* callback function is called to inform the Solution Policy DLL that a volume has been discovered or a network share has been mapped. This callback allows the Solution Policy DLL to tell FESF whether it should process files on the volume/share.

## Syntax

```
bool  
POL_ATTACH_VOLUME(  
    _In_ GUID* VolumeGUID,  
    _In_ FE_POLICY_VOLUME_INFORMATION* VolumeInformation  
);
```

## Parameters

### *VolumeGUID [in]*

The volume's GUID. This will be the GUID uniquely identifying the volume, or one of the FESF-defined GUIDS: **FE\_SHADOW\_VOLUME\_GUID**, **FE\_NETWORK\_GUID**, or **FE\_NO\_VOLUME\_GUID\_AVAILABLE**. Note that starting in FESF V2 this parameter is input to the Solution Policy DLL only.

### *VolumeInformation [in]*

A pointer to an **FESF\_VOLUME\_INFORMATION** structure that contains information about the volume or share that's been discovered.

## Return value

If the DLL returns **TRUE**, then FESF will process files on this volume.

## Remarks

This callback is optional. If it is not provided, FESF will process files on the volume/share.

If *PolAttachVolume* returns **TRUE**, FESF will process future files accessed on the volume/share. The Solution Policy DLL's *PolGetPolicyNewFile* will be called for new files created on the volume/share, and *PolGetPolicyExistingFile* will be called when an FESF encrypted file is opened on the volume/share.

If *PolAttachVolume* returns **FALSE**, FESF will not process future files accesses on the volume/share. As a result, the Solution Policy DLL's callback will not be involved for any operations on the volume/share. In this case, FESF either performs no subsequent processing at all on the volume or reduces its processing on the volume to the absolute minimum possible. This callback provides the Solution Policy DLL with a powerful mechanism to reduce the number of callbacks it will receive, by allowing it to eliminate entire volumes from policy consideration.

The values passed as arguments into this structure are as provided to the FESF kernel-mode filter components by the Windows Filter Manager. See, for example, the Microsoft documentation for the **PFLT\_INSTANCE\_SETUP\_CALLBACK** callback function.

Note that *PolAttachVolume* is not called for partitions that are not processed by Windows, nor is it called for any OEM partitions, the system recovery partition(s), UEFI boot partition(s) or for file systems that are not supported by FESF (such as CDFS or UDF). By design FESF does not process file accesses on these volumes.

When a volume that has not yet been presented to the Solution Policy is accessed, and the Solution Policy DLL is available, FESF V2 will present the volume to the Solution Policy DLL's **PolAttachVolume** callback (and "remember" that the Solution Policy DLL has been presented this volume).

When a volume that has not yet been presented to the Solution Policy is accessed (such as during system startup), and the Solution Policy DLL is not running, FESF V2 will allow access to raw files and will deny access to existing FESF encrypted files.

*PolAttachVolume* is called only once (during the course of system operation) for each discovered volume. FESF "remembers" whether a given volume (including volumes on network shares) has been presented to the Solution Policy DLL. Thus, if Fe2Policy and the Solution Policy DLL are restarted, it will not receive notification of the volumes that had previously been presented to the *PolAttachVolume* callback.

If *PolAttachVolume* is implemented by the Solution Policy DLL, the volume must not be accessed by the Solution until *PolAttachVolume* has successfully returned and Windows has completed volume mount processing. Thus, the Solution Policy DLL cannot call functions such as **GetVolumeInformation** or **CreateFile** for any file targeting the volume being mounted from within its *PolAttachVolume* callback. Prematurely accessing the volume being mounted from within the Solution Policy DLL will lead to a Windows deadlock. This is a Windows operating system restriction.

## Changes From FESF V1

In FESF V1, the Solution Policy DLL could optionally return a GUID of its choosing by overwriting the value passed in the VolumeGUID parameter. Starting in FESF V2.0 this option is no longer supported. If the Solution Policy DLL attempts to return a changed value in the VolumeGUID parameter in FESF V2.0 or later, the Policy Service logs an error and changes the return value to "FALSE" thus causing FESF to NOT process files on the volume.

Starting in FESF V2, this callback is called for all supported volumes and network shares on the system, even those that were discovered by Windows before the Solution Policy DLL was started. FESF tracks whether a given volume/share has been presented to the Solution Policy DLL since the system was started. If it has not, FESF will present the volume/share to the Solution Policy DLL when a new CreateFile file operation is performed on the volume/share. This allows the Solution Policy DLL to indicate whether FESF should support encryption and decryption operations on the volume.

## See Also

## Requirements

Software version	FESF V1.6 (or later) / Changes as noted in FESF 2.0 (and later)
Library	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolFreeHeader callback function

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

## Syntax

```
POL_FREE_HEADER PolFreeHeader;  
  
VOID  
PolFreeHeader(  
    _In_ PVOID PolHeaderData,  
    _In_ DWORD PolHeaderDataSize  
)
```

## Parameters

*PolHeaderData* [in]

A pointer to a Solution Header Data area to be returned that was previously allocated by the Solution Policy DLL.

*PolHeaderDataSize* [in]

The size, in bytes, of the Solution Header Data area.

## Return value

(none)

## Remarks

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

Solution Policy DLLs must implement this callback function.

## 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	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolFreeKey callback function

A Solution Policy DLL's *PolFreeKey* callback function is called to enable the Solution 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 Solution Policy DLL.

*PolKeySize* [in]

The size, in bytes, of the key storage area.

## Return value

(none)

## Remarks

A Solution Policy DLL's *PolFreeKey* callback function is called by FESF to allow the Solution Policy DLL to deallocate space that it previously allocated for storage of key data information. This key buffer was provided to FESF by the Solution 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 Solution Header Data (which is presumably not security sensitive) and key information (which is presumably sensitive from a security standpoint). In most Solution Policy DLL implementations *PolFreeKey* would overwrite the key storage area with random data before freeing it.

Solution Policy DLLs must implement this callback function.

## 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	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolGetKeyFromHeader callback function

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

### *ThreadId* [in]

The identifier of the thread creating the file.

### *PolHeaderData* [in]

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

### *PolHeaderDataSize* [in]

The size, in bytes, of the Solution 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 Solution Policy DLL in the **FE2\_POLICY\_CONFIG** structure passed to **FePolicySetConfiguration**.

*PolKey [out]*

A pointer to a storage area allocated by the Solution Policy DLL containing encryption key data. FESF will pass this key data to the CNG cryptographic algorithm provider indicated by *Algorithm*. FESF calls the Solution 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 Solution Policy DLL defined context value that will be passed to the Solution Policy DLL's *PolReportLastHandleClosed* callback function. This parameter is optional and may be **NULL**.

If this value is specified *and* the file already has a *CleanupInfo* registered then you will be called back at *PolReportLastHandleClosed* with the previous value to *CleanupInfo*.

## 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**. See the section Returning Failure from a Solution Policy DLL

## Remarks

All Solution Policy DLLs must implement this callback function.

A Solution 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 crypto 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 Solution Policy DLL derives the encryption algorithm and key data from the provided thread, header and **FE\_POLICY\_PATH\_INFORMATION** (including requestor SID) for the file.

*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.

Solution 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. Als.

## 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	Fe2Policy.lib
Header	Fe2PolDllApi.h



# PolGetKeyNewFile callback function

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

## Syntax

```
POL_GET_KEY_NEW_FILE_EX PolGetKeyNewFile;

bool
PolGetKeyNewFile(
    _In_ FE_POLICY_PATH_INFORMATION *PolicyPathInfo,
    _In_ DWORD ThreadId,
    _In_ PVOID Context,
    _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 and the SID of the requestor.

### *ThreadId* [in]

The identifier of the thread creating the file.

### *Context*[in]

Caller-supplied value previously returned as output from the *PolGetPolicyNewFile* callback.

### *PolHeaderData* [out]

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

### *PolHeaderDataSize* [out]

The size, in bytes, of the Solution 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 Solution Policy DLL in the **FE2\_POLICY\_CONFIG** structure passed to **FePolicySetConfiguration**.

*PolKey [out]*

A pointer to a storage area allocated by the Solution Policy DLL containing encryption key data. FESF will pass this key data to the CNG cryptographic algorithm provider indicated by *Algorithm*. FESF calls the Solution 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 Solution Policy DLL defined context value that will be passed to the Solution Policy DLL's *PolReportLastHandleClosed* callback function. This parameter is optional and may be **NULL**.

If this value is specified *and* the file already has a *CleanupInfo* registered then you will be called back at *PolReportLastHandleClosed* with the previous value to *CleanupInfo*

## 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 Solution Policy DLLs must implement this callback function.

A Solution 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 Solution Header Data returned in *PolHeaderData* from this function as well as certain FESF control information in the file. FESF stores the Solution Header exactly as it is provided by the Solution Policy DLL. FESF does not process or encrypt this data.

After 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 Solution Header Data will be retrieved from the file and returned to the Solution Policy DLL at its *PolGetKeyFromHeader* callback function. Given this Solution Header Data, the Solution 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.

*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.

Solution 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 Solution Policy DLL Callbacks](#).

## 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	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolGetLockRounding callback function

When present, a Solution Policy DLL's *PolGetLockRounding* callback function is called to find the “lock rounding” which should be associated with all accessors to a file prior to the locking request being sent to the remote server. See Remarks.

## Syntax

```
FESF_LOCK_ROUNDING
POL_GET_LOCK_ROUNDING(
    _In_ FE_POLICY_PATH_INFORMATION* PolicyPathInfo
);
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE\_POLICY\_PATH\_INFORMATION** structure describing the file being created and the SID of the requestor. The *VolumeGuid* member will always be **FE\_NETWORK\_GUID**.

## Return value

The Solution Policy DLL must return one of these values:

- **FESF\_LOCK\_ROUNDING::InsideFileOnly** to indicate that rounding should only be applied within the current length of the file. This is the default value.
- **FESF\_LOCK\_ROUNDING::AlwaysOn** to indicate that all ranges are always rounded.
- **FESF\_LOCK\_ROUNDING::Off** to indicate that rounding should never be applied for this file.

## Remarks

This function is called to set the rounding applied to LockFile operations prior to the request being passed across to remote file servers. All IO from FESF is done via the cache and hence is aligned to page boundaries. It is therefore usual to round all byte range lock requests out to the page boundary – this ensures that IO will always be accepted by the remote server.

Some applications are known to use byte range locks not to protect against application conflict, but rather to signal between remote copies of the application. Typically, such applications will lock a single byte, usually well beyond the end of file, but occasionally within the file. In this situation, the rounding of byte range locks stops the application from functioning whilst suppressing the lock rounding does not impede function (since it is not being used to arbitrate access to regions of the file).

The default rounding applied by FESF is to round to the page boundaries up to the end of file (plus a small margin). This has been found to be the best compromise. This default behavior can be overridden on a system wide basis by a registry setting.

This API allows control of the rounding on a per file basis. So, for instance, files only accessed by Outlook, which is known to use lock rounding within the file in certain versions, can be marked as “never round”.

## See Also

## Requirements

Software version	FESF V1.8 (or later)
Library	Fe2Policy.lib
Header	Fe2PoIDllApi.h

# PolGetPolicyDirectoryListing callback function

When present, allows the Solution Policy DLL to determine whether the sizes returned in a given directory listing reflect the “raw” or “corrected” sizes for FESF encrypted files.

## 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 Solution 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 **directory enumeration** operation to fail with an error.

## Remarks

When 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, 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 Solution 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.

The raw (uncorrected) size of the file is the size of the file on disk, including storage for the Solution Header and FESF control metadata. The corrected size of the file is the size of the file's data, which is the size that the file's data would be if it were not FESF encrypted.

This function is optional. If it is not implemented, then all directory enumerations will see the encrypted size (as if **FE\_POLICY\_ENCRYPT\_DECRYPT** was always returned).

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 directory 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 Solution Policy DLL. For more information, see [FESF Policy Caching](#).

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

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, OSR was surprised to find a number of applications that did this. Two specific examples are the xcopy command, and the Microsoft OneDrive application. However, we wouldn't be surprised if other programs such as certain backup applications behaved similarly.

## See Also

## Requirements

Software version	FESF V1.1 (or later)
Library	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolGetPolicyExistingFile callback function

A Solution 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 and the SID of the requestor.

### *ThreadId* [in]

The identifier of the thread opening the file.

### *PolHeaderData* [in]

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

### *PolHeaderDataSize* [in]

The size, in bytes, of the Solution 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 Solution 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\_POLICY\_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\_POLICY\_RAW**, read and write operations on the file will be performed on the data as provided. That is, no transparent encryption or decryption of data will 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. Solution Policy DLLs should return **FE\_POLICY\_FAIL** from their *PolGetPolicyDirectoryListing* callback only when absolutely necessary. For more information see [Returning Failure from Solution Policy DLL Callbacks](#) elsewhere in this document.

## Remarks

All Solution Policy DLLs must implement this callback function.

FESF calls a Solution 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. In this case, FESF uses the cached policy, and the Solution Policy DLL is not called.
- When an encrypted file is opened transactionally (such as the result of a thread calling **CreateFileTransacted**). These opens are denied by FESF with STATUS\_TRANSACTIONAL\_CONFLICT without the Solution Policy DLL being called. Note that transactions have only ever been supported by NTFS, are not supported by ReFS or FAT, and have been deprecated by Microsoft for many years.

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

FESF calls a Solution 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. The security principal (SID) under which the requestor is running is provided in the *FE\_POLICY\_PATH\_INFORMATION* structure.

If the **FE\_POLICY\_PATH\_FILE\_DEHYDRATED** flag is set in the *PolicyFlags* field of the provided *PolicyPathInfo* structure, the file being accessed is locally “dehydrated” and encrypted. In this context, “dehydrated” means that the file’s data is located in the cloud and only a placeholder representing the file is stored locally on disk.

Each cloud provider behaves differently, and you will need to understand the precise behavior and the consequences of recalling dehydrated files. However, in general, if your Solution intends to handle files that are “unencrypted in the cloud” you should never return raw to opens that have the **FE\_POLICY\_PATH\_FILE\_DEHYDRATED** flag set.

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. The FESF Policy Cache can be flushed at any time by the Solution 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 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, *PolGetPolicyNewFile* is called when an existing zero length file is encountered (including a file that is superseded as part of being opened).

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\_POLICY\_ENCRYPT\_DECRYPT** and the other open is granted **FE\_POLICY\_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 Solution 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.

## 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	Fe2Policy.lib
Header	Fe2PolDllApi.h

# PolGetPolicyNewFile callback function

A Solution 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,
    _Out_ PVOID *Context
)
```

## Parameters

*PolicyPathInfo* [in]

A pointer to an FESF allocated **FE\_POLICY\_PATH\_INFORMATION** structure describing the file being created and the SID of the requestor.

*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:

<b>FILE_READ_DATA</b> 0x001	The right to read the file data.
<b>FILE_READ_DATA</b> 0x002	The right to write data to the file.
<b>FILE_APPEND_DATA</b> 0x004	The right to append data to the file.
<b>FILE_READ_EA</b> 0x008	The right to read extended file attributes.

**FILE\_WRITE\_EA**

0x010

The right to write extended file attributes.

**FILE\_EXECUTE**

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**

0x080

The right to read file attributes.

**FILE\_WRITE\_ATTRIBUTES**

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**

0x000

An existing file was deleted and a new file was created in its place.

**POL\_CREATE\_ACTION\_OPENED**

0x001

An existing file was opened.

**POL\_CREATE\_ACTION\_CREATED**

0x002

A new file was created.

**POL\_CREATE\_ACTION\_OVERWRITTEN**

0x003

An existing file was overwritten

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

*Context [out]*

A Solution Policy DLL determined value for FESF to pass to the matching *PolGetKeyNewFile* callback (which will immediately follow if this function returns **FE\_POLICY\_ENCRYPT\_DECRYPT**).

**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\_POLICY\_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* may return the enumeration value **FE\_POLICY\_FAIL**. This will cause the thread's **CreateFile** operation to fail with an error. Solution Policy DLLs should return **FE\_POLICY\_FAIL** from their

*PolGetPolicyDirectoryListing* callback only when absolutely necessary. For more information see [Returning Failure from Solution Policy DLL Callbacks](#) elsewhere in this document.

## Remarks

All Solution Policy DLLs must implement this callback function.

A Solution 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. The Security Identifier (SID) of the account under which the thread is running is passed to the Solution Policy DLL in the *FE\_POLICY\_PATH\_INFORMATION* structure. Given the SID, the Solution can get the associated username (for example, one way to do this in C++ is by calling the Windows function **LookupAccountSid**).

Whenever *PolGetPolicyNewFile* returns **FE\_POLICY\_ENCRYPT\_DECRYPT**, FESF will call the Solution Policy DLL at its *PolGetKeyNewFile* callback function to retrieve the Algorithm ID, Key, and Solution Header Data for the file.

It is important to understand that FESF considers all 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\_POLICY\_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.

## 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	Fe2Policy.lib
Header	Fe2PoIDllApi.h

# PolReportFileInconsistent callback function

[Obsolete and unavailable as of FESF V2.0]

# PolReportLastHandleClosed callback function

A Solution Policy DLL's *PolReportLastHandleClosed* callback function is called to inform the Solution 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 Solution 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 Solution Policy DLL's *PolReportLastHandleClosed* callback function is called by FESF to inform the Solution Policy DLL that the last handle has been closed. This callback allows the Solution Policy DLL to trigger additional processing of the file. Since the last handle has been closed, a Solution that attempts to open the file in the context of this callback will usually succeed without encountering a sharing violation. Note, however, that subsequent opens before the callback is called make sharing violation still possible, just unlikely. Hence your Solution may need to be constructed to gracefully handle this possibility, for instance, by queuing the work to be done later.

It is important to realize that *PolReportLastHandleClosed* callback will be called even if the *CleanupInfo* value was established up by a previous instance of your Solution Policy DLL (such as when Fe2Policy is restarted). Therefore, Solution Policy DLLs should never use this parameter to pass a pointer to a DLL structure (because that pointer will no longer be valid if Fe2Policy exits, and your Solution Policy DLL is reloaded).

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

## See Also



## Requirements

Software version	FESF V1 (or later)
Library	Fe2Policy.lib
Header	Fe2PoIDllApi.h

# PolUnInit callback function

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

Solution Policy DLLs need not implement this callback function.

## See Also

## Requirements

Software version	FESF V1 (or later)
Library	Fe2Policy.lib
Header	Fe2PolDllApi.h

## 11 FESF Policy Function Reference

---

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

# FePolSetConfiguration function

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

## Syntax

```
DWORD  
FePolSetConfiguration(  
    _In_ FE2_POLICY_CONFIG * Configuration  
)
```

## Parameters

*Configuration [in]*

A pointer to a **FE2\_POLICY\_CONFIG** structure that has been filled-in by the Solution 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 **FE2\_POLICY\_CONFIG** structure, **ERROR\_INVALID\_DATA** is returned.

## Remarks

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

Any structures that are passed in to **FePolSetConfiguration** may be freed as soon as **FePolSetConfiguration** returns to the caller.

## See Also

For an illustration of how to set up the **FE2\_POLICY\_CONFIG** structure and calling **FePolicySetConfiguration** within the Solution Policy DLL's **PolicyDllInit** callback function, see SampPolicy.cpp in the Sample Solution.

## 12 FesfUtil2 Function Reference

---

The FESFUtil2 DLL is a helper library, designed to assist Client Solutions in performing various FESF utility operations. FESFUtil2 are designed for use when FESF is installed and running on the system. Note FesfUtil2 exports both native C and C++ interfaces as described below.

### 12.1 Using FesfUtil2

#### 12.1.1 C versus C++ API

The FesfUtil2 APIs are available in two forms: C and C++. These two APIs are entirely interchangeable and interoperable but, to avoid confusion, we recommend that you choose and stick with one set of APIs.

In the FesfUtil2 Library there is a C Language function for each C++ Function. Given a C++ function named **FESFUtil2Function** the equivalent function for C will be named **FESFUtil2Function\_C**. The documentation below describes the C++ function but also provides the C prototype.

All the C++ functions report errors by throwing an exception of type FEU2Exception. The FEU2Exception structure contains a Win32 error code (and, if applicable, an NTSTATUS value) indicating the error encountered. FEU2Exception also provides methods for rendering strings that describe the error.

Note the following regarding the C Language functions:

- Every C Language function returns a Win32 Error Code. If the function succeeds, the function returns **ERROR\_SUCCESS**.
- When a C++ function returns a value, the C equivalent returns the value in one or more [out] parameters. In these situations, the documentation only refers to the return value.
- Strings that are passed as input must be null terminated.
- Where a buffer and length is passed for the return of a string value, on input the buffer size is specified in bytes and must include the maximum string length that can fit in the buffer, **including the trailing null character** (2 bytes long since all strings are wide-character strings).  
On return, the buffer length is set by FesfUtil2 to the length (in bytes) of the returned string **NOT including the terminating null character**.
- If a buffer and length are passed as input to an FesfUtil2 function, and the indicated buffer is not large enough, FesfUtil2 will return **ERROR\_MORE\_DATA** and the minimum required buffer length will be provided on output in the buffer length parameter.
- If a C++ function returns a **FESF\_UTIL2\_SOLUTION\_HEADER**, then the C function has three extra parameters: A pointer to the buffer into which the header is to be returned, a pointer to a DWORD containing the length of the buffer on input, and returning the size of the header on output, and a pointer to a DWORD in which is returned the maximum possible length of the header in the file without extension.

#### 12.1.2 Security Context of called functions

Note that FesfUtil2 is a directly callable DLL and thus will run in the context of, and therefore with the security credentials of, the calling application. When a function requires the caller to have specific privileges, those privileges are documented in the function's description. When a given function requires specific privileges, and those privileges are not enabled when the function is called, FesfUtil2 will attempt to temporarily enable the required privileges for the duration of the function call (and disable those privileges before returning). If FesfUtil2 is not able to enable the required privileges, an error is returned.

If you are migrating a Solution from the legacy FesfDs Service (the "OSR FESF Data Storage Service"), recall that FesfDs ran with full administrator privileges and therefore, in some cases, could be used by applications with lesser privileges

to perform what might otherwise be privileged operations. Calling applications will now need to hold all necessary privileges to perform any functions that are called.

### 12.1.3 Static and Dynamic Libraries

To allow Solution Developers to select the Microsoft C/C++ run time library option (static or DLL-based) of their choice, FesFUtil2 is shipped as both a static and a dynamic (DLL) library. The Static library lib files are called "FesFUtil2\_S" (Release) and "FesFUtil2\_SD" (Debug). The Dynamic library lib and dll files are both called FesFUtil2.

### 12.1.4 Tracing and Debugging with FesFUtil2

The Debug and Release versions of FesFUtil2 allow you to specify both the debug/tracing level (volume) and locations for output of debug/trace messages (referred to as the debug "mode"). The mode options are: to the console (for console mode programs), to the debugger, to a file, via message box pop-up, or any combination of these simultaneously. While all these modes are supported for Debug builds of the FesFUtil2 libraries, the Release build of the FesFUtil2 libraries only supports logging to the debugger or to a file.

The trace "level" and trace "mode" (that is, where trace output is shown) are controlled via Registry entries. The active level and mode are maintained separately for each application using FesFUtil2 and are read each time an application using the debug version of FesFUtil2 is started.

The values are:

HKLM\Software\OSR\

FEU2DebugMode : REG\_DWORD      Where the trace/debug output appears

FEU2TraceLevel:    REG\_DWORD      The trace/debug verbosity.

Supported Values:

FEU2DebugMode:

0x001	Write messages to a log file
0x002	Write messages to the debugger
0x004	Display message, as it occurs, in a pop-up message box (DEBUG only)
0x100	Write messages to the command window (DEBUG only)

These flags can be or'ed together in any combination.

**IMPORTANT:** Note that for 32-bit applications (32-bit version of FesFUtil2.dll or 32-bit apps that are statically built with FesFUtil2) the registry key will be redirected to the 32-bit specific registry. So, while HKLM\Software\OSR may have specific values set, 32-bit apps will not see these values. You need to set the values for 32-bit apps under:

\HKLM\SOFTWARE\WOW6432Node\OSR\

When messages are written to a log file, that file is %TMP%\FesFUtil2\_<PID>.log. The file is created if it is not already present, and messages are appended to the file if it already exists. The file is opened when an application with logging is started and is closed then the application exits. In Release builds, the file is opened "WRITE\_THROUGH."

Recall that files created in the %TMP% directory on Windows are never automatically deleted. Thus, some thought is needed when enabling logging to a file. Also, because file names use the PID of the application, it's possible to quickly create a lot of small (potentially empty) trace files. This happens, for example, when you enable logging at the trace

level and run one of the sample tools (like SampDir). Every time you run one of the tools, a new log file will be created.

When messages are displayed in a pop-up message box, one message box is popped for each debug message, and it must be dismissed before the app continues. This option can be used from either console mode or windowed programs. It's most useful when trying to track down a rare issue.

When messages are displayed to the command window, they are written to stdout. This option is only useful if the application has a command window. Setting this flag for a GUI app has no effect.

FEU2TraceLevel:

0x001	Trace-level
0x002	Warning-level
0x003	Error-level
0x004	Critical-level

You select one value from the list above, and trace/debug messages at the specified level or higher will be generated (using the mode specified by the FEU2DebugMode parameter). For Release builds, we force any value less than or equal to Trace-level to Warning-level (to prevent a situation where the system is barely responsive in the field by enabling Trace-level debugging, which can be voluminous).

# FesfUtil2FixFileTag Function

Corrects the file size tagging "hint" on an FESF encrypted file. See the remarks section for more information.

## Syntax (C++)

```
void  
FesfUtil2FixFileTag(std::wstring_view PathToFix);
```

## Syntax (C)

```
DWORD  
FesfUtil2FixFileTag_C(  
    [in] LPCWSTR PathToFix);
```

## Parameters

### *PathToFix*

The fully qualified path to an FESF encrypted file to be updated.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

FESF uses a file's allocation size as a "hint" as to whether the file is encrypted by FESF. This allows FESF to identify files as being FESF encrypted without having to open each file and look for FESF meta-data. In rare cases, if a file is backed-up or otherwise altered, that file's length hint can become "broken" causing FESF to no longer recognize an FESF encrypted file as being encrypted. This function fixes the tagging for that file.

Requires the caller to have SeRestorePrivilege. If the privilege is not already activated, it will be activated by the function and deactivated before return.



# FesfUtil2GetExecutablePathForThreadId Function

Given a Thread ID, returns a fully qualified path to the thread's executable image.

## Syntax (C++)

```
std::wstring  
FesfUtil2GetExecutablePathForThreadId(uint32_t ThreadId);
```

## Syntax (C)

```
DWORD  
FesfUtil2GetExecutablePathForThreadId_C(  
    [in] DWORD ThreadId,  
    [out] LPWSTR ReturnedPath,  
    [out] DWORD* PathBufferSize);
```

## Parameters

*ThreadId*

Thread ID for which to locate the executable.

## Return value

A fully qualified path to the Thread Id's executable image.

## Throws

Throws an FEU2Exception if an error is encountered.

## Replaces

IFesfUtil::GetExecutablePathForThreadId

# FesfUtil2GetFileSize Function

Returns the actual size (that is, the size including any FESF private meta-data and the Solution Policy Header).

## Syntax (C++)

```
uint64_t  
FesfUtil2GetFileSize(std::wstring_view FileToCheck);
```

## Syntax (C)

```
DWORD FesfUtil2GetFileSize_C(  
    [in] LPCWSTR FileToCheck,  
    [out] DWORD64* TrueFileSize);
```

## Parameters

### *FileToCheck*

Fully qualified path name for the file.

## Return value

The size of the file, in bytes.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

The "actual" or "true" size of a file that is FESF encrypted is different from the "corrected" file size. The "corrected" size is the size that indicates how much user-data is stored in the file and most closely approximates the size that the file would be if it were NOT encrypted. The "actual" size (returned by this function) is the size that includes all the FESF private meta-data as well as the Solution Policy Header. This size most accurately accounts for the amount of space occupied by the file on disk.

No special privileges are required, though the caller must have read access to the file being queried if the file is local and both read and write access to the file being queried if the file is on the network. Network files require exclusive access.

## Replaces

IFesfDs::GetTrueSize and IFesfUtil::GetTrueFileSize

# FesfUtil2GetFullyQualifiedPath Function

Given a Volume GUID and a path relative to that Volume GUID, returns a fully qualified path specification for the file, as well as an indication as to whether the file is likely to be accessed via the network.

## Syntax (C++)

```
std::pair<std::wstring, bool>  
FesfUtil2GetFullyQualifiedPath(  
    GUID VolumeGUID,  
    std::wstring_view RelativePath);
```

## Syntax (C)

```
DWORD  
FesfUtil2GetFullyQualifiedPath_C(  
    [in] GUID    VolumeGUID,  
    [in] LPCWSTR RelativePath,  
    [out] BOOL*   IsNetworkPath,  
    [out] PWSTR   ReturnedPath,  
    [inout] DWORD* PathBufferSize));
```

## Parameters

### *VolumeGUID*

A Windows Volume ID, or the FESF-reserved values FE\_NETWORK\_GUID, FE\_SHADOW\_VOLUME\_GUID, or GUID\_NULL.

### *RelativePath*

A file path specification relative to the *VolumeGUID*.

## Return value

A string containing a fully qualified path for the file, plus a bool indicating whether the returned path represents a path on the network.

## Throws

Throws an FEU2Exception if an error is encountered.

## Replaces

IFesfUtil::GetFullyQualifiedLocalPath

# FesfUtil2GetSidForThreadId Function (deprecated)

Retrieves the Security Identifier SID under which a given thread is running. Note that this function is deprecated. See the Remarks for further information.

## Syntax (C++)

```
std::wstring  
FesfUtil2GetSidForThreadId(uint32_t ThreadId);
```

## Syntax (C)

```
FesfUtil2GetSidForThreadId_C(  
    [in] DWORD ThreadId,  
    [out] LPWSTR SidString,  
    [inout] DWORD* SidBufferSize);
```

## Parameters

### *ThreadId*

The ID of a currently active thread for which to query the SID.

## Return value

A string containing the SID under which *ThreadId* is running.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

May fail for security reasons when called for certain Windows "protected processes". This is part of the Windows architecture and cannot be bypassed.

*Starting in FESF V2.0 this function is deprecated*, in favor of using the SID value that's passed to the Solution Policy DLL as part of the FE\_POLICY\_PATH\_INFORMATION structure. In all versions of FESF, there is an extremely rare case where the SID returned by this function can be wrong. This case only occurs for kernel-mode callers, and when the requestor has performed a file operation after attaching a thread to another process by calling KeStackAttachProcess. Thus, in FESF V2.0 the *RequestorSID* field was added to the **FE\_POLICY\_PATH\_INFORMATION** structure that's passed to various Solution Policy DLL callbacks. The value passed in that parameter is always accurate.

OSR has no plans to immediately remove this function because the error case is so rare. However, developers should consider moving their Solutions to use the SID passed in FE\_POLICY\_PATH\_INFORMATION whenever this is possible.

## Replaces

IFesfUtil::GetSidForThreadId

# FesfUtil2GetUniversalFilePath Function

Given a file path, which may only be valid within a given session, returns a path which is usable across all sessions on the current system.

## Syntax (C++)

```
std::wstring  
FesfUtil2GetUniversalFilePath(std::wstring_view FilePath);
```

## Syntax (C)

```
DWORD  
FesfUtil2GetUniversalFilePath_C(  
    [in] LPCWSTR FilePath,  
    [out] LPWSTR ReturnedPath,  
    [inout] DWORD* PathBufferSize);
```

## Parameters

*FilePath*

A file path to query.

## Return value

A fully qualified ("UNC") file path that is valid in all sessions.

## Throws

Throws an FEU2Exception if an error is encountered.

# FesfUtil2GetVersion Function

Returns the current version of the FesfUtil2 library.

## Syntax (C++)

```
std::pair<uint32_t, uint32_t>  
FesfUtil2GetVersion();
```

## Syntax (C)

```
DWORD  
FesfUtil2GetVersion_C(  
    [out] DWORD* VersionMajor,  
    [out] DWORD* VersionMinor);
```

## Return value

Two integer values, the first of which represents the FesfUtil2 major version and the second represents the FesfUtil2 minor version.

## Throws

Throws an FEU2Exception if an error is encountered.

# FesfUtil2IsFileFesfEncrypted Function

Check to see if a file is in FESF Encrypted format.

## Syntax (C++)

```
bool  
FesfUtil2IsFileFesfEncrypted(std::wstring_view FileToCheck);
```

## Syntax (C)

```
DWORD  
FesfUtil2IsFileFesfEncrypted_C(  
    [in] LPCWSTR FileToCheck,  
    [out] BOOL*   FileIsEncrypted);
```

## Parameters

*FileToCheck*

Fully qualified path name for the file.

## Return value

Returns 'true' if the file is FESF encrypted, 'false' otherwise.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

No special privileges are required, though the caller must have read access to the file being queried.

## Replaces

IFesfDs::IsFileEncrypted and IFesfUtil::IsFileEncrypted

# FesfUtil2IsThreadIdInSid Function

Determines if the thread indicated by *ThreadId* is in the group indicated by *GroupSidString*.

## Syntax (C++)

```
bool  
FesfUtil2IsThreadIdInSid(uint32_t ThreadId,  
                          std::wstring_view GroupSidString);
```

## Syntax (C)

```
DWORD  
FesfUtil2IsThreadIdInSid_C(  
    [in]_DWORD   ThreadId,  
    [in] LPCWSTR GroupSidString,  
    [out] BOOL*   ThreadIsInSid);
```

## Parameters

*ThreadId*

Thread ID of the thread to check.

*GroupSidString*

String containing a SID to check for membership.

## Return value

Returns 'true' if *ThreadId* is a member of group *GroupSidString*, 'false' otherwise.

## Replaces

IFesfUtil::IsThreadIdInSid



# FesfUtil2PurgePolicyCache Function (file variant)

Purges data stored in the FESF Policy Cache for the specified file.

## Syntax (C++)

```
void  
FesfUtil2PurgePolicyCache(std::wstring_view PathToPurge);
```

## Syntax (C)

```
DWORD  
FesfUtil2PurgePolicyCacheForFile_C([in] LPCWSTR PathToPurge);
```

## Parameters

### *PathToPurge*

Fully qualified path for the file to purge from cache. A fully specified file name is required. Wild cards in the file specification are not supported.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

This function causes FESF to remove all references to a given file from the FESF Policy Cache. There may be some delay between the time this function is called and when the Policy Cache is completely purged for the specified file.

## Replaces

IFesfUtil::PurgePolicyCacheFile

# FesfUtil2PurgePolicyCache Function (thread variant)

Purges data stored in the FESF Policy Cache for the process identified by the specified thread.

## Syntax (C++)

```
void  
FesfUtil2PurgePolicyCache(uint32_t ThreadIdToPurge);
```

## Syntax (C)

```
DWORD  
FesfUtil2PurgePolicyCacheForThreadId_C([in] DWORD ThreadIdToPurge);
```

## Parameters

### *ThreadIdToPurge*

ID of a thread that identifies the process to be purged. If *ThreadIdToPurge* is zero, the FESF Policy Cache is purged for all processes in the system that are active at the time of the call.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

This function 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 being completely purged. Even though this function takes a Thread ID, the FESF Policy Cache is purged for all threads in the process that owns the specified thread.

If the *ThreadIdToPurge* parameter is passed as zero, the FESF Policy Cache is purged for all threads and all processes.

## Replaces

IFesfUtil::PurgePolicyCacheThread

# FesfUtil2ReadHeaderExclusive Function

Reads and returns the Solution Header for an FESF encrypted file. Opens the file for exclusive access, and requires both read and write access to the file.

## Syntax (C++)

```
FESF_UTIL2_SOLUTION_HEADER  
FesfUtil2ReadHeaderExclusive(std::wstring_view FileToRead);
```

## Syntax (C)

```
DWORD  
FesfUtil2ReadHeaderExclusive_C(  
    [in] LPCWSTR FileToRead,  
    [out] PVOID HeaderBuffer,  
    [inout] DWORD* HeaderLength)
```

## Parameters

*FileToRead*

Fully qualified path name for the file.

## Return value

Returns an [FESF\\_UTIL2\\_SOLUTION\\_HEADER](#) instance that describes and contains the file's Solution Policy Header.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

Requires the caller to have SeRestorePrivilege. If the privilege is not already activated, it will be activated by the function and deactivated before return.

## Replaces

IFesfDs::ReadHeader

# FesfUtil2ReadHeaderUnsafe Function

Reads and returns the Solution Header for an FESF encrypted file. Opens the file "shared" and therefore presents a risk of other programs simultaneously accessing (and possibly updating) the file's header directly via a call to FesfUtil2 or via FESF. When possible, use the function FesfUtil2ReadHeaderExclusive in preference to this function.

## Syntax (C++)

```
FESF_UTIL2_SOLUTION_HEADER  
FesfUtil2ReadHeaderUnsafe(std::wstring_view FileToRead);
```

## Syntax (C)

```
DWORD  
FesfUtil2ReadHeaderUnsafe_C(  
    [in] LPCWSTR FileToRead,  
    [out] PVOID HeaderBuffer,  
    [inout] DWORD* HeaderLength,  
    [Out] DWORD* MaxHeaderSize)
```

## Parameters

*FileToRead*

Fully qualified path name for the file.

## Return value

Returns an [FESF\\_UTIL2\\_SOLUTION\\_HEADER](#) instance that describes and contains the file's Solution Policy Header.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

Requires the caller to have SeRestorePrivilege. If the privilege is not already activated, it will be activated by the function and deactivated before return.

This function only supports local files. It can **NOT** be used to read the header of a file that is accessed via the network.

## Replaces

IFesfUtil2::ReadHeaderUnsafe and IFesfUtil2::ReadHeaderUnsafeFQP

# FesfUtil2SetPolicyCacheState Function

Allows Solutions to globally enable or disable the FESF policy cache.

## Syntax (C++)

```
bool  
FesfUtil2SetPolicyCacheState(bool DesiredState);
```

## Syntax (C)

```
BOOLEAN  
FesfUtil2SetPolicyCacheState_C(  
    [in] BOOLEAN DesiredState);
```

## Parameters

### *DesiredState*

Set to "true" to enable the FESF global policy cache; Set to "false" to disable the cache.

## Return value

A bool value indicating the previous state of the policy cache. 'True' indicates that prior to this call, the policy cache was enabled; 'false' indicates that the policy cache was previously disabled.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

On success, when *DesiredState* is 'false' this function will globally purge the policy cache (removing all entries from the cache).

This function is new as of FESF V2.0

# FesfUtil2UpdateHeaderExclusive Function

Replaces the existing Solution Header on the file indicated by *FileToUpdate* with the Solution Header provided in *NewHeader*. Opens the file for exclusive access and requires both read and write access to the file.

## Syntax (C++)

```
void  
FesfUtil2UpdateHeaderExclusive(std::wstring_view FileToUpdate,  
                               const FESF_UTIL2_SOLUTION_HEADER &NewHeader);
```

## Syntax (C)

```
DWORD  
FesfUtil2UpdateHeaderExclusive_C(  
    [in] LPCWSTR FileToUpdate,  
    [inout] PVOID HeaderBuffer,  
    [in] DWORD HeaderLength);
```

## Parameters

### *FileToUpdate*

Fully qualified path of file to operate on.

### *NewHeader*

Header to be substituted for the existing header.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

Requires the caller to have SeRestorePrivilege. If the privilege is not already activated, it will be activated by the function and deactivated before return.

NewHeader must fit in the existing Solution Header space. That is, it must not be larger than the MaxLength size returned when the existing Solution Header is read. To write a header that is larger than the existing maximum size, use FesfUtil2UpdateHeaderExclusiveWithExtension.

## Replaces

IFesfDs::UpdateHeader

# FesfUtil2UpdateHeaderExclusiveWithExtension Function

Updates the existing Solution Header of *FileToUpdate*. Opens the file for exclusive access; the caller must have both read and write access to the file. *NewHeader* may be any (non-zero) size, including larger or smaller than the existing header.

## Syntax (C++)

```
void
FesfUtil2UpdateHeaderExclusiveWithExtension(
    std::wstring_view      FileToUpdate,
    const FESF_UTIL2_SOLUTION_HEADER &NewHeader,
    const std::wstring_view &      BackupFileTag = L"");
```

## Syntax (C)

```
DWORD
FesfUtil2UpdateHeaderExclusiveWithExtension_C(
    [in] LPCWSTR FileToUpdate,
    [in] PVOID HeaderBuffer,
    [in] DWORD HeaderLength,
    [inopt] LPCWSTR BackupFileTag);
```

## Parameters

### *FileToUpdate*

Fully qualified path name of file to receive the new header.

### *NewHeader*

Valid instance of a new header to be substituted for the existing one.

### *BackupFileTag*

A tag to use in forming the backup file name. Defaults to an empty string. See remarks section for more information.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

Requires the caller to have SeRestorePrivilege, SeTakeOwnershipPrivilege, and SeSecurityPrivilege on the system where the file is located. If these privileges are not already activated, they will be activated by the function and deactivated before return. Requires the caller to have both read and write access to the file.

This function allows a file to be updated with a Solution Header that is larger than that which will fit in the current file. To accomplish this, an entirely new file is created with the new header. The name of the new file is the same as the current file, but with the suffix “\_FESF\_TEMP\_<random-number>” added, where “<random\_number>” is a random string of up to 10 decimal digits. The raw data from the original file is copied to the new file, the original file's characteristics are propagated to the new file using the Win32 function **ReplaceFile**, and the original file's security attributes (DACL and SACL) are propagated to the new file.

If a non-zero-length *BackupFileTag* is specified by the caller, the name of the backup file will incorporate that tag in the name. If a name is not specified, the tag "FESF\_BACKUP" will be used. The backup file name is used by the Win32 **ReplaceFile** (which is called by this function). In addition, if the caller specifies a *BackupFileTag*, the backup file will be retained on return, even when this function is successful. If a *BackupFileTag* is not specified, and this function is successful, the backup file is deleted before returning to the caller.

The name of the backup file is formed by pre-pending the backup tag (either the one passed-in by the caller or the default value) to the filename portion of the fully qualified file path. The backup tag is separated from the original file name by a "\_" character. For example:

FileToUpdate: C:\Fred\bob.txt

Default backup file name: C:\Fred\FESF\_BACKUP\_bob.txt

This scheme was devised to avoid clashing with the name of the original file and to make it more likely to preserve the original file's short name.

Note that the size of the original file name and extension must be short enough to allow:

- a) The new file to be created with the “\_FESF\_TEMP\_<random-number>” suffix.
- b) The backup file to be created with the backup tag, which defaults to “FESF\_BACKUP\_”.

Note that, while fully qualified paths on Windows may be up to 32767 characters in length, the maximum length of a file name plus file extension on NTFS is 255 characters.

## Replaces

IFesfDs::UpdateHeaderWithExtension



# FesfUtil2UpdateHeaderUnsafe Function

Replaces the existing Solution Header on the file indicated by *FileToUpdate* with the Solution Header provided in *NewHeader*. Opens the file "shared" and thus presents a risk of other programs simultaneously accessing (and possibly updating) the file's header directly via a call to FesfUtil2 or via other ordinary operations by FESF. When possible, we *strongly* urge using the function FesfUtil2UpdateHeaderExclusive in preference to this function.

## Syntax (C++)

```
void
FesfUtil2UpdateHeaderUnsafe(std::wstring_view FileToUpdate,
                           const FESF_UTIL2_SOLUTION_HEADER &NewHeader);
```

## Syntax (C)

```
DWORD
FesfUtil2UpdateHeaderUnsafe_C(
    [in] LPCWSTR FileToUpdate,
    [in] PVOID HeaderBuffer,
    [in] DWORD HeaderLength);
```

## Parameters

### *FileToUpdate*

Fully qualified path of file to operate on.

### *NewHeader*

Header to be substituted for the existing header.

## Throws

Throws an FEU2Exception if an error is encountered.

## Remarks

Requires the caller to have SeRestorePrivilege. If the privilege is not already activated, it will be activated by the function and deactivated before return.

*NewHeader* must fit in the existing Solution Header space. That is, it must not be larger than the *MaxLength* size returned when the existing Solution Header is read. To write a header that is larger than the existing maximum size, use FesfUtil2UpdateHeaderExclusiveWithExtension.

This function only supports local files. It can NOT be used to read the header of a file that is accessed via the network.

## Replaces

IFesfUtil2::UpdateHeaderUnsafe and IFesfUtil2::UpdateHeaderUnsafeFQP

## 12.2 FesfUtil2 Classes & Structures

This section describes the classes and structures of FesfUtil2.

# FESF\_UTIL2\_SOLUTION\_HEADER

This structure is the representation of a FESF solution header.

## Methods

### *Constructors*

```
FESF_UTIL2_SOLUTION_HEADER(uint32_t HeaderSize = 0);
```

Create a new Solution Header.

```
FESF_UTIL2_SOLUTION_HEADER(const FESF_UTIL2_SOLUTION_HEADER &Rhs);
```

Create a new Solution Header as a COPY of the provided input.

```
FESF_UTIL2_SOLUTION_HEADER(FESF_UTIL2_SOLUTION_HEADER &&Rhs);
```

Create a new Solution Header with the identical contents (as a MOVE)

### *SetCurrentLength*

```
void SetCurrentLength(const uint32_t Length);
```

Sets the size of the header. See notes below.

### *CurrentLength*

```
uint32_t CurrentLength();
```

Gets the size of the header. See notes below.

### *MaxLength*

```
uint32_t MaxLength();
```

Gets the maximum size of the header. See notes below.

### *Get*

```
void* Get();
```

Returns a pointer to the header data. See notes below.

### *IsValid*

```
bool IsValid();
```

Returns whether the header is valid.

### *ReallocateBuffer*

```
void ReallocateBuffer(uint32_t NewSize);
```

Increases the maximum size available for the header.

## Notes

The Solution Header itself is accessed using the `.Get` method. So, for example, if you successfully call `FesfUtil2ReadHeader` as follows:

```
auto theHeader = FesfUtil2ReadHeader("C:\\fred\\myfile.txt");
```

The Solution Header that's been read from the file and returned to you can be accessed by:

```
theHeader.Get()
```

In terms of lengths, note that a Solution Header has two "length" values:

### *.CurrentLength()*

This is the currently used/valid length of the solution header.

When a header is copied or written, *CurrentLength* is the number of bytes written. When a header is read, *CurrentLength* is the number of valid bytes in the header.

### *.MaxLength()*

This is the maximum header size that can be written to the associated file without requiring the header to be extended.

When an `FESF_UTIL2_SOLUTION_HEADER` is instantiated or reallocated (with the `ReallocateBuffer` method), the *HeaderSize* provided is used to size the allocated buffer and both *MaxLength* and *CurrentLength* are initially set to this size. The *CurrentLength* can be subsequently updated to less than *MaxLength* by calling `SetCurrentLength()`.

When a header is read, the header buffer is allocated with *MaxLength* bytes, and the size of the Solution Header Data that was actually read is indicated by *CurrentLength*.

# FESFException

This structure is used by FesfUtil2 to describe errors encountered during processing.

The primary purpose of this structure is to convey either a Win32 or, more rarely, an internal NTSTATUS value and its meaning to the caller.

## Methods

### Win32Error

Returns a DWORD containing the Win32 Error Value indicating the error that FesfUtil2 encountered during processing.

### NtStatus

Returns a DWORD containing a native, Windows kernel-mode, NTSTATUS value indicating the error that FesfUtil2 encountered during processing.

### IsNativeError

Returns a bool indicating whether the error value being returned is an NTSTATUS (IsNativeError is set to true) or is a Win32 error (IsNativeError is set to false).

### GetMessageString

Returns an std::string with the text of the error message, suitable for direct output or logging. The Win32 error code and (if applicable) the NTSTATUS value are also displayed as part of the output.

### what

Returns a pointer to a null-terminated const char string (NOT a wide-character string) with the text of the error message, suitable for direct output or logging. The Win32 error code and (if applicable) the NTSTATUS value are also displayed as part of the output. The only difference between the values returned by this and the GetMessageString() methods is the format of the output. The message text is identical.

## 13 FESF Stand-Alone Library Function Reference

---

The functions in this section are implemented by the FESF V2 Stand-Alone library (Fe2Sa.lib).

### 13.1 About the Stand-Alone Library

The Stand-Alone Library is a supported part of FESF that is provided and maintained by OSR. As such, the code for the library itself is not to be changed by licensees. On the other hand, the **Fe2SaCyrpt** utility is part of the FESF Sample Solution Components and may be used, customized, or adapted by FESF licensees in any way they see fit.

The FESF Stand Along Library was created to provide licensees a supported mechanism to create and access FESF encrypted files on systems where FESF is not installed. It is not designed to be high performance. Additionally, the Library is not guaranteed to be inherently thread safe.

### 13.2 About the FE2Sa Functions

Probably THE 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). Using the FesfSa functions on systems where FESF is active will result in undefined behaviors, including file corruption.

#### **PLEASE READ THIS: Important Note on the Future Direction of the FESF Stand-Alone Library**

As described above, the only supported use of the FESF Stand-Alone Library is on systems where FESF is not installed.

As part of ongoing efforts to update the FESF architecture, OSR plans to change the FESF Stand-Alone Library so that its functions will fail (returning an error) on systems on which FESF is running.

This should have no impact on your product because the Library was never supported for use on systems where FESF was installed. If, however, your Solution DID make use of the FESF Stand-Alone Library on systems where FESF was running, this will eventually no longer be possible. We recommend using FesfUtil2 or some other approach. Please only ever use the FESF Stand-Alone Library on systems in which FESF is known not to be running.

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 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).

Please refer to the **Fe2SaCrypt** utility, which is part of the unsupported FESF Sample Solution Components (the Sample.sln Solution) for an example of implementing an FESF-compatible encryption scheme, including building and using the ESSIV.

Starting with FESF V2, **Fe2SaCrypt** uses the open source **SymCrypt** library. This library is developed and supported by Microsoft and is the underlying engine used by the Windows CNG Library for both user-mode and kernel-mode encryption. *FESF does not ship with any **SymCrypt** components*, but they are available on GitHub as both [source code](#) and [pre-built binaries](#). **SymCrypt** supports Windows, Linux, and macOS. We have tested **Fe2SaCrypt** using the new **Fe2Sa** library and **SymCrypt** (AES/CBC 128) on both Windows and Linux (Ubuntu).

The Fe2SaCrypt source code has both build and implementation comments that can help guide your adaptation of the example to your specific use.

### 13.2.1 Building Fe2SaCrypt on Windows

The build procedures for this **Fe2SaCrypt** (for both Windows and Linux) assume the **SymCrypt** files have been downloaded and are available in the standard FESF sample user source code tree.

To build **Fe2SaCrypt** on Windows, download the zip archive appropriate for the architecture that you're building (ARM64 or AMD64), extract the **inc** and **dll** directories, and locate those directories in the FESF user source tree under `src\user\SymCrypt\Windows`.

When properly unzipped, you should have the following with the SymCrypt components:

```
src\user\SymCrypt\Windows\<arch>\dll\  
src\user\SymCrypt\Windows\<arch>\inc\
```

Where <arch> is either "x64" or "arm64". Note that the build procedure as provided does not look in configuration-specified subdirectories to differentiate between Release and Debug builds.

### 13.2.2 Building Fe2SaCrypt on Linux

To build **Fe2SaCrypt** on Linux, download the tar.gz archive appropriate for the architecture that you're building (ARM64 or AMD64). Copy it to the `src/user/SymCrypt` directory on your Linux system (note that, as of this writing, the **SymCrypt** library tgz contains files that are symlinks that cannot be copied correctly via Windows File Explorer). Under the SymCrypt directory, create a directory named "Linux". Extract the contents of the SymCrypt inc and lib directories from the SymCrypt tar.gz file using the following command:

```
tar -xzf x.tar.gz -C Linux ./lib ./inc
```

You should get the following with the **SymCrypt** components:

```
src/user/SymCrypt/Linux/lib/  
src/user/SymCrypt/Linux/inc/
```

Then build using the supplied makefile.

Note that you will need the contents of the Linux/lib directory in the same directory as your executable (fe2sacrypt) when you run.

### 13.2.3 About Building Fe2SaCrypt on Non-Windows Platforms

Please keep in mind that Fe2SaCrypt is supplied by OSR purely as an example. It is part of the FESF **unsupported** Sample Solution Components. While OSR is not able to provide assistance for building or using Fe2SaCrypt on non-Windows platforms, we are always pleased to hear about your experiences (both good and bad) with any of the sample utilities.

# DecryptCallback function

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 both Windows and Linux platforms, the **errno** variable is 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 Fe2Sa Functions* in this document for the description of these issues. Note that the **Fe2SaCrypt** sample utility demonstrates how to implement a compatible IV, encryption, and blocking scheme using the **SymCrypt** library and AES-128/CBC.

## Requirements

Software version	FESF Version 1 (added)
Supported FESF State	FESF Not Installed ONLY
Windows Library	Fe2Sa.lib
Linux Library	Libfe2sa.a

# EncryptCallback function

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 WriteEncrypted
)
```

## 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**.

*WriteEncrypted [in]*

If **TRUE**, the contents of the Data buffer must be encrypted before storing the output. If **FALSE**, Data must be written without modification. See the Remarks section for more information.

## Return value

Returns **TRUE** if the function successfully processes all the data, **FALSE** otherwise.

For both Windows and 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 *WriteEncrypted* 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 *WriteEncrypted* is **TRUE** and the data supplied in *StreamData* is not an integer multiple of the *CipherBlockSize* the application receiving the callback is responsible for padding the data appropriately before performing the encryption operation.

If *WriteEncrypted* 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 Fe2Sa Functions* in this document for the description of these issues. Note that the **Fe2SaCrypt** sample utility demonstrates how to implement a compatible IV, encryption, and blocking scheme using the **SymCrypt** library and AES-128/CBC.

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.

Requirements

Software version	FESF Version 1 (added)
Supported FESF State	FESF Not Installed ONLY
Windows Library	Fe2Sa.lib
Linux Library	Libfe2sa.a

# 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 *CallbackRoutine*.

## Return value

Returns **TRUE** if the function successfully processes all the data, **FALSE** otherwise.

For both Linux and Windows 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.

## Requirements

Software version	FESF Version 1 (added)
Supported FESF State	FESF Not Installed ONLY
Windows Library	Fe2Sa.lib

Linux Library	Libfe2sa.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 both Windows and 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 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.

Requirements

Software version	FESF Version 1 (added)
Supported FESF State	FESF Not Installed ONLY
Windows Library	Fe2Sa.lib
Linux Library	Libfe2sa.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 and 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.

## Requirements

Software version	FESF Version 1 (added)
Supported FESF State	FESF Not Installed ONLY
Windows Library	Fe2Sa.lib
Linux Library	Libfe2sa.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 and 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 **errno** is set to be **-E2BIG**.  
In all other error cases *BytesRead* is set to be **0xFFFFFFFF**.

## Requirements

Software version	FESF Version 1.1 (added)
Supported FESF State	FESF Not Installed ONLY

Windows Library	Fe2Sa.lib
Linux Library	Libfe2sa.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 both Windows and 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	Fe2Sa.lib
Linux Library	Libfe2sa.a

## 14 FESF Policy Data Structures

---

This section describes the FESF Policy data structures.

# FE2\_POLICY\_ALGORITHM structure

The **FE2\_POLICY\_ALGORITHM** structure specifies the encryption algorithm and associated information. This information is used by FESF to call CNG.

## Syntax

```
typedef struct _FE2_POLICY_ALGORITHM {

    //
    // A string that uniquely identifies this algorithm to the FESF
    // Service.
    //
    LPCWSTR PolUniqueAlgorithmId;

    //
    // Passed as the pszAlgId parameter to
    // BCryptOpenAlgorithmProvider
    //
    LPCWSTR CNGAlgorithmIdentifier;

    //
    // Passed as the pszImplementation parameter to
    // BCryptOpenAlgorithmProvider
    //
    LPCWSTR CNGAlgorithmImplementation;
    bool UseESSIV;
    FE2_POLICY_ALGORITHM_VERSION FesfPolicyAlgorithmVersion;
    FE2_POLICY_CRYPT0_TYPE CryptoType;
    FE2_POLICY_KEY_LENGTH KeyLength;

} FE2_POLICY_ALGORITHM;
```

## Members

### *PolUniqueAlgorithmId*

A pointer to a Solution Policy DLL defined null-terminated constant wide character string, that will be used to identify this particular algorithm and specified properties. The Solution Policy DLL provides this string as an output from its *PolGetKeyNewFile* and *PolGetKeyFromHeader* callback functions.

### *CNGAlgorithmIdentifier*

This value is ignored except for custom cryptographic algorithm providers, where it is pszAlgId parameter to the CNG function BCryptOpenAlgorithmProvider.

*CNGAlgorithmImplementation*

This value is ignored except for custom cryptographic algorithm providers, where it is pszImplementation parameter to the CNG function BCryptOpenAlgorithmProvider.

*UseESSIV*

This value is ignored except for custom cryptographic algorithm providers. See Remarks for more information.

*FesfPolicyAlgorithmVersion*

Specify **FE2\_POLICY\_ALGORITHM\_VERSION::VERSION2**. No other values are currently supported.

*CryptoType*

Specify one of the following values to select the cryptographic algorithm to use:

**FE2\_POLICY\_CRYPTO\_TYPE::AES\_CBC\_ESSIV** – This selects the “standard” FESF encryption algorithm that has been historically used by almost all FESF licensees.

**FE2\_POLICY\_CRYPTO\_TYPE::AES\_XTS** – This selects AES with XTS mode; *This option is not currently supported, but will be implemented in a later release of FESF.*

**FE2\_POLICY\_CRYPTO\_TYPE::CUSTOM** – This selects a custom-developed crypto algorithm provider.

*KeyLength*

Specify one of the following values:

**FE2\_POLICY\_KEY\_LENGTH::BITS\_128** (this value is not valid with **AES\_XTS**)

**FE2\_POLICY\_KEY\_LENGTH::BITS\_256**

**FE2\_POLICY\_KEY\_LENGTH::BITS\_512** (this value is not valid with **AES\_CBC\_ESSIV**)

For **AES\_XTS**, this length specifies the size of the key PAIR that’s being used. The pair of keys is provided by the Solution Policy DLL as a single concatenated value. Thus, for XTS mode, specifying **BITS\_256** indicates that two (concatenated) 128 bit keys will be supplied by the Solution Policy DLL in a single 256 bit buffer.

## Remarks

Important notes for developers/users of custom cryptographic algorithm providers (only):

- FESF assumes the cryptographic block size is 16 bytes (as it is in AES)
- FESF uses an encryption block size of 256 bytes; This is the interval at which we compute a new offset and (if specified) ESSIV.
- When using a custom cryptographic algorithm, if you set **UseESSIV** to true FESF will compute the 16-byte ESSIV (as described elsewhere in this document) and pass a pointer to it to your crypto algorithm (as the pbIV and cbIV) on calls to **BCryptEncrypt** and **BCryptDecrypt**. If you set **UseESSIV** to false FESF will pass a pointer to a 16-byte buffer containing the byte offset of the block being encrypted to your crypto algorithm (as the pbIV and cbIV) on calls to **BCryptEncrypt** and **BCryptDecrypt**. These two options should allow you to perform whatever type of encryption you need.

## See Also

## Requirements

Software version	FESF V1 (and later)
Header	Fe2PolDIIApi.h

# FE2\_POLICY\_CONFIG structure

The **FE2\_POLICY\_CONFIG** structure specifies the selected configuration options and callbacks for the Solution Policy DLL.

## Syntax

```
typedef struct _FE2_POLICY_CONFIG {
    DWORD VersionMajor;

    DWORD VersionMinor;

    DWORD Length;

    struct {
        bool Enable;
        bool mbf1;
        bool mbf2;
        bool mbf3;
    } AccessCache;

    struct {
        bool res1;
        bool mbf1;
        bool mbf2;
        bool mbf3;
    } NetworkBehavior;

    //
    // Maximum size of the Solution Header used by your
    // Solution.
    //
    DWORD    SolutionHeaderMaximumSize;

    POL_GET_POLICY_NEW_FILE          *PolGetPolicyNewFile;
    POL_GET_KEY_NEW_FILE_EX          *PolGetKeyNewFile;
    POL_GET_POLICY_EXISTING_FILE     *PolGetPolicyExistingFile;
    POL_GET_POLICY_DIRECTORY_LISTING *PolGetPolicyDirectoryListing;
    POL_GET_KEY_FROM_HEADER           *PolGetKeyFromHeader;

    //
    // Optional. If not specified, all renames are approved.
    //
```



```

POL_APPROVE_RENAME          *PolApproveRename;

//
// Optional. If not specified, all requests to create links
// are approved.
//
POL_APPROVE_CREATE_LINK     *PolApproveCreateLink;

//
// Optional. If not specified, no action is taken.
//
POL_REPORT_LAST_HANDLE_CLOSED *PolReportLastHandleClosed;

//
// Optional. If not specified, FESF attaches to all volumes
//
POL_ATTACH_VOLUME          *PolAttachVolume;

//
// Optional - Network locking
//
POL_GET_LOCK_ROUNDING      *PolGetLockRounding;

//
// Required
//
POL_FREE_HEADER            *PolFreeHeader;
POL_FREE_KEY               *PolFreeKey;

//
// Optional. If not specified, no action is taken
//
POL_UNINIT                 *PolUnInit;

//
// Virtualization Filters we need to ignore and their Virtualized Directories
//
DWORD                      VirtualizationFilterCount;
LPCWSTR                    *VirtualizationFilters;
LPCWSTR                    *VirtualizationDirs;

//
// The crypto algorithm(s) to use

```

```
//
// Note that there is a pre-configured maximum number of algos that
// are supported by FESF V2. Changing this requires changing FESF V2
// kernel-mode code to match.
//
DWORD AlgorithmsCount;
FE2_POLICY_ALGORITHM Algorithm[FE2_CRYPT_MAX_SUPPORTED_ALGOS];

} FE2_POLICY_CONFIG;
```

## Members

### *VersionMajor*

The major version of the FESF Policy API supported by the Solution Policy DLL. This must be **FE\_POLICY\_VERSION\_MAJOR**.

### *VersionMinor*

The minor version of the FESF Policy API supported by the Solution Policy DLL. This must be **FE\_POLICY\_VERSION\_MINOR**.

### *Length*

The length in bytes of the **FE2\_POLICY\_CONFIG** structure.

### *AccessCache*

#### *Enable*

Set to **TRUE** to enable FESF Policy Caching. Otherwise, set to **FALSE**. Note that after setting this to an initial value via this structure, a Solution can change FESF's Policy Caching behavior using the FESF support function `Fe2UtilSetPolicyCacheState`.

### *NetworkBehavior*

(Currently unused; Set to zero.)

### *SolutionHeaderMaximumSize*

Maximum size (in bytes) of your Solution Header Data that FESF should expect to encounter. `Fe2Policy` uses the value you provide here to size the buffers it pre-allocates for communications with the kernel-mode FESF components. Setting this value to be arbitrarily large will waste (real, physical) memory on customer machines. Setting this value too small will result in a larger Solution Header you provide being rejected (resulting in unpredictable application behavior). See elsewhere in this document for a more detailed description of how this value is used.

### *PolGetPolicyNewFile*

A pointer to the Solution Policy DLL's `PolGetPolicyNewFile` callback function.

*PolGetKeyNewFile*

A pointer to the Solution Policy DLL's *PolGetKeyNewFile* callback function.

*PolGetPolicyExistingFile*

A pointer to the Solution Policy DLL's *PolGetPolicyExistingFile* callback function.

*PolGetPolicyDirectoryListing*

A pointer to the Solution Policy DLL's *PolGetPolicyDirectoryListing* callback function

*PolGetKeyFromHeader*

A pointer to the Solution Policy DLL's *PolGetKeyFromHeader* callback function.

*PolApproveRename*

A pointer to the Solution Policy DLL's *PolApproveRename* callback function.

*PolApproveCreateLink*

A pointer to the Solution Policy DLL's *PolApproveCreateLink* callback function.

*PolReportLastHandleClosed*

A pointer to the Solution Policy DLL's *PolReportLastHandleClosed* callback function

*PolAttachVolume*

An optional pointer to the Solution Policy DLL's *PolAttachVolume* callback function

*PolGetLockRounding*

An optional pointer to the Solution Policy DLL's *PolGetLockRounding* callback function

*PolFreeHeader*

A pointer to the Solution Policy DLL's *PolFreeHeader* callback function.

*PolFreeKey*

A pointer to the Solution Policy DLL's *PolFreeKey* callback function.

*PolUnInit*

A pointer to the Solution Policy DLL's *PolUnInit* callback function.

*AlgorithmsCount*

A count of entries in the vector of the Algorithms member of this structure.

*Algorithm*

A vector of **FE2\_POLICY\_ALGORITHM** structures, each of which describes an encryption algorithm that the Solution Policy DLL will use. Up to **FE2\_CRYPT\_MAX\_SUPPORTED\_ALGOS** (which is currently 6, but is subject to change) may be defined by a Solution Policy DLL.

## Remarks

## See Also

## Requirements

Software version	FESF V1 (or later)
Header	Fe2PoIDllApi.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 RequestorSID;
    LPCWSTR RelativePath;
    DWORD   PathFlags;
    UUID     VolumeGuid;
    Union {
        LPCWSTR ServerAndShare;
        LPCWSTR ShadowVolumeName;
    };
} FE_POLICY_PATH_INFORMATION;
```

## Members

### *RequestorSID*

A string containing the Security ID (SID) associated with the security principal performing the file operation.

### *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:

<b>FE_POLICY_PATH_NAME_NOT_NORMALIZED</b>	Indicates that the path information provided has NOT been normalized in format. This is a rare occurrence and relates only to some specific network operations.
<b>FE_POLICY_PATH_TARGET_NAME_INVALID</b>	Indicates that the target name could not be established (typically, as a result of a rename across a directory junction point). This flag is only set for <i>PolApproveCreateLink</i> and <i>PolApproveRename</i> callbacks.
<b>FE_POLICY_PATH_NAME_BYPASS</b>	Indicates that the file name (source or target) has been detected as being in a “bypass” region. A bypass region is one in which file opens are always raw (because of interoperability issues). For example, files used to boot

the system might be in a bypass region. The Solution Policy DLL is never involved in Policy determination for BYPASS files, but a rename of a file from or to a “bypass” region may be of interest to a Solution Policy DLL.

**FE\_POLICY\_PATH\_FILE\_DEHYDRATED**

Indicates that a file is being recalled from a Cloud Storage provider (such as OneDrive). This flag will only be set for the *PolGetPolicyExistingFile* callback. See the description of that callback for more information.

*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**

The SID value passed in the *RequestorSID* field should be used in preference to retrieving the SID from the thread by calling **FesfUtil2GetSidForThread** (qv), thereby avoiding a rare situation in which the SID returned by FesfUtil2 is wrong. This field is populated for create operations, including **PolGetPolicyNewFile**, **PolGetPolicyExistingFile**, and **PolApproveTransactedOpen**.

**See Also**

**Requirements**

Software version	FESF V1 (or later)
Header	Fe2PolDllApi.h

# FE\_POLICY\_VOLUME\_INFORMATION structure

The **FE\_POLICY\_VOLUME\_INFORMATION** structure information about a volume or network share that is being presented to the Solution Policy DLL in the *AttachVolume* callback.

## Syntax

```
typedef struct _FE_POLICY_VOLUME_INFORMATION {

    ///
    /// @brief The volumes (kernel-mode) device name
    /// Typically of the form "\\Device\\HarddiskVolume5"
    ///
    PCWSTR DeviceName;

    ///
    /// @brief Volume's Device Object flags, copied directly from
    /// the DEVICE_OBJECT structure's Flags field.
    ///
    DWORD DeviceFlags;

    ///
    /// @brief Volume's Device Object characteristics, copied
    /// directly from the DEVICE_OBJECT's Characteristics
    /// field.
    ///
    DWORD DeviceCharacteristics;

    ///
    /// @brief Volume Device Object device type, copied directly
    /// from the DEVICE_OBJECT's DeviceType field.
    ///
    DWORD VolumeDeviceType;

    ///
    /// @brief Volume's file system type, as described by the
    /// FLT_FILESYSTEM_TYPE enumeration
    ///
    DWORD VolumeFilesystemType;

    ///
    /// @brief The Volume Label
    ///
    ///

```

```
LPCWSTR VolumeLabel;  
  
} FE_POLICY_VOLUME_INFORMATION;
```

## Members

### *DeviceName*

Pointer to a wide-character string containing the volume's (kernel) device name – typically of the form “\Device\HarddiskVolume5”. For network shares this name is of the form “\\Server\Share”.

### *DeviceFlags*

The volume's Device Object flags, copied directly from the **DEVICE\_OBJECT** structure's *Flags* field. For network shares, this field will be zero and should be disregarded.

### *DeviceCharacteristics*

The volume's Device Object characteristics, copied directly from the **DEVICE\_OBJECT** structure's *Characteristics* field. For network shares, this field will be zero and should be disregarded.

### *VolumeDeviceType*

The volume's Device Object device type, copied directly from the **DEVICE\_OBJECT** structure's *DeviceType* field.

### *VolumeFilesystemType*

The volume's file system type, as described by the **FLT\_FILESYSTEM\_TYPE** enumeration.

### *VolumeLabel*

Pointer to a wide-character string containing the volume's label, as read from the disk. Note this field is not used for network shares or shadow volumes.



## 15 FESF and Support for Cloud Storage

Since its earliest design phases, one of the primary goals for FESF has been “interoperability.” Webster’s New World College Dictionary (4<sup>th</sup> Edition) as cited by collinsdictionary.com defines interoperability as

*the ability of a system or component to function effectively with other systems or components.*

Obviously, we (OSR and FESF Licensees) only control one side of the interoperability equation. This is important to recognize, because there’s literally no guarantee at all about the behaviors, quality, or even the basic engineering soundness of the components with which FESF might seek to interoperate.

In this document, we’ll briefly explore the vectors of interoperability. We’ll then describe OSR’s policies with respect to FESF interoperability, and our stance on support for interoperability issues. Because they are so important, we’ll focus separately on some of the issues raised by Cloud Storage Providers like OneDrive, Dropbox, and Box Sync.

### 15.1 Why Interoperability Is Complex

FESF for Windows is implemented as a set of Windows File System Minifilters. These filters instantiate above Windows File Systems and intercept traffic on its way to and from those File Systems. More specifically, FESF uses an exceptionally complex type of Minifilter referred to as an [Isolation Minifilter](#). This is what allows FESF to provide simultaneous decrypted and raw views of a file.

When we think about interoperability here at OSR, we typically consider four distinct areas of operation:

- The ability of ordinary (user-mode) applications to perform file operations “as usual”, using the file systems that FESF is filtering. This includes FESF providing encryption/decryption of file data transparently, based on policies defined by the Solution. Note that this doesn’t only include supporting read and write, however; It also includes other aspects, such as support for corrected file lengths in directory and attribute queries.
- The ability of FESF to work across multiple, different, versions of the Windows operating system.
- The ability of FESF to interoperate with standard Windows components, including Microsoft-developed File Systems and File System Minifilters.
- The ability of FESF to interoperate with non-Microsoft developed components that implement features and functionality similar to file systems, implement actual unique file systems, or that impact a standard Microsoft file system. These are most typically Windows File System Filters and File System Minifilters developed by vendors. This category includes, for example, non-Microsoft developed antivirus products and non-Microsoft developed cloud storage products.

File System interoperability is sufficiently complex that the only way to know if a given set of products is interoperable is to try them and see if they all work together. A proof point for this is that for the past 15 years or so Microsoft has hosted regular “Plugfests.” According to the official Microsoft invitation, the goal of these events is:

*[T]o help you prepare your file system minifilter, network filter, or boot encryption driver for the next version of Windows by performing interoperability testing with other products.*

An OSR engineering team regularly attends these events along with teams from more than 50 other software vendors worldwide and Microsoft product teams. During the five days that Plugfest lasts, our team works collaboratively with other teams to ensure our products work together and with the latest version of Windows. When interoperability problems are discovered, engineering teams from both companies are present to identify and solve them.

Still, serious interoperability problems between Windows and File System components built by first-tier vendors are common. For example, consider the [serious problems](#) caused by the April 2019 Monthly Rollup (KB4493472) for many big AntiVirus vendors (including Sophos, McAfee, Avira, Avast, ArcaBit and others).

Why is File System interoperability so complex? What can we do about it? Those are the topics we'll address in the remainder of this document.

#### 15.1.1 Unwritten Rules and Many Permutations

It may surprise you to learn that most of the rules for creating interoperable File System Minifilters are unwritten, ridiculously complex, and are constantly changing. This even includes application interoperability. The result is FESF doesn't achieve any of these categories of interoperability automatically. There's work involved for every, single, release.

This is true, even though we here at OSR helped originate the filtering approach used by modern File System Filters and Minifilters. We pioneered the development of "same stack" filtering back in the 1990's. Microsoft officially endorsed this approach starting in the mid-2000's, and it is the approach that they currently recommend. The result has been dramatically increased interoperability, across the whole ecosystem.

Even when "same stack" filtering is used, however, the work involved to achieve interoperability can be daunting. While an application reading an encrypted file and being presented with decrypted data might seem reasonably straight-forward, supporting even this simple operation can involve stunningly complex interactions with the I/O Subsystem, the Memory Manager, the Cache Manager, and the underlying file system.

The case above becomes even more complicated when the differing behavior of various Windows versions are added into the mix.

Then consider the differences that occur due to the (multiple different versions of) underlying Microsoft file systems.

Then add into the mix the multiple Microsoft-developed File System Minifilters that ship with each version of Windows.

And finally, don't forget to account for the unique behaviors introduced by any coexisting third-party products, such as antivirus filters, backup filters, license managers, hierarchical storage subsystems, etc., each of which can vary from release to release.

#### 15.1.2 Ever Changing Rules of the Game

Another issue that makes OS version interoperability and Microsoft interoperability so difficult is that the rules, and the interfaces, constantly change. And not just in small ways.

For example, Microsoft's Cloud Filter File System Minifilter suddenly started using an Extra Create Parameter with the ID GUID\_ECP\_ATOMIC\_CREATE. This ECP was introduced in Windows 10 RS1 or RS2 but wasn't used by OneDrive until sometime during RS4. And even when they started using it, the feature was never meaningfully documented. So, as FESF developers, we're forced to wait to see how this feature is used in the real world. The result? An interoperability problem until we can "catch up" to Microsoft's implementation changes.

#### 15.1.3 Third Parties Doing Whatever They Want

File system Minifilters *can* be very simple to write. However, File System Minifilters are particularly difficult to write *correctly*. This is because many of the interactions that they require with the file systems are entirely undocumented and only learnable through experience. So, not even every Windows kernel mode software engineer has a grasp of the complexity of this field.

Because of the degree of specialization that's inherent in the Windows File System space, it is a mistake to assume that software packages authored by large, successful, and/or reputable companies are reliable, well-written, or automatically likely to be interoperable. Corporate success or even engineering competence in other specialties is absolutely no indicator of interoperability in this space. There's a cloud storage solution built by a major vendor that is built upon a framework that is well-known within the Windows File System community to be "less than reliable." This software doesn't work with FESF, and unless the underlying framework is dramatically changed (or the software is re-written) it will never work properly with FESF.

When you combine these factors with the fact that there is no "quality gate" that a third party must pass to be allowed to ship their product, the result is a lot of **very** poorly implemented Minifilters. These Minifilters show their poor engineering by breaking things in the system. Like FESF.

#### 15.1.4 Applications Do Very Silly Things

There is no limit to the strange behaviors that applications undertake. And, when they're doing regular reads and writes to a local disk, and there's no encryption or decryption (Isolation) involved, those behaviors rarely have much noticeable impact. But add Isolation into the mix? There are certain behaviors that result in problems that are simply irreconcilable.

During the FESF V1.5 cycle, we received reports of an application that, quite literally, opens a file *over the network* for sequential access and writes 3 bytes at a time. Does that sound silly to you? It sure sounds silly to us. Want even sillier? This application also opens the file with shared read permissions, allowing OTHER applications to simultaneously open the file for read access.

So, the user writes 3 bytes. We need to encrypt an entire block, so we need to read it from the server. There's somebody on another system somewhere who also opens the file, but for read access and they need their updates, so we write the encrypted block back to the server. And we do this every time the application writes three bytes. This access pattern results in "less than optimal" (cough) performance by FESF. Surprised that this app gets bad performance? We weren't. We still don't know how to fix this.

#### 15.1.5 Windows Versions Constantly Change

A final factor that adds to the complexity of the interoperability problem is that Windows itself is almost never the same thing twice. Because of "Windows as a service" the version of Windows 11 24H2 (for example) that you download today is probably not the same as the version of Windows 11 24H2 that you downloaded last week. And the changes aren't necessarily subtle. We've seen major changes in how, for example, OneDrive works within a given version of Windows.

This makes it difficult for all of us to reproduce problems. It can make it hard for you to repro problems reported by your customer sites; It can be hard for us to repro problems that you report to us.

We see this, particularly, when preparing for major upgrades of Windows... say, from 1809 to 19H1. Thankfully, Microsoft has at least returned to the practice of designating releases as "Release Candidates." But even once a release becomes "final" it continues to change. This is what makes planning to support these upgrades so difficult. We can test with, for example, the release of 20H2 that's available from Microsoft today. But there's no guarantee at all that this release will behave the same as next week's release of 20H2 – And this is true both before and after 20H2 is officially released.

#### 15.1.6 Third Party Software Versions Constantly Change

Given that "Test in Production" is all the rage these days, constant version churn isn't restricted just to Microsoft components. We've seen third party code, both applications and File System Minifilters, change both dramatically

and sometime frequently. The fact that the name of a third-party component hasn't changed doesn't necessarily mean its internal components or implementation mechanisms are the same.

Of course, this just serves to further complicate interoperability and repro scenarios.

Prior to the release of a recent version of FESF, we did very thorough testing with a specific Cloud Storage Provider. We were happy to know that FESF was fully interoperable. Imagine our surprise, some months later, when we got a problem report saying that FESF didn't work *at all* with this same package. We did some quick exploration and discovered (surprise!) that the mechanism the product used to store and retrieve files from the cloud had completely changed.

## 15.2 OSR's Approach to Interoperability for FESF

At this point, I hope we've made clear the fact that when it comes to File System Minifilters, doing good engineering and playing by the rules is not enough to ensure interoperability. There are too many variables at play, those variables are constantly changing, "the rules" aren't clear and they're not well-understood by all the players. Plus, there exist all manner of File System Minifilters and applications that behave pretty much however they want.

The result is that interoperability problems will show up. They are an inherent part of developing products in this space. We just must live with this reality. That's the hard truth.

So, is all lost? What do we do?

After 30 years of working in this space, here at OSR we've come up with some strategies. And we use these strategies to guide our FESF design, development, and support activities.

### 15.2.1 Recognize Interoperability Problems as a Fact of Life

The first step to solving a problem is recognizing that it exists. We've just covered this. There will always be interoperability problems. These problems are not (usually) the result of bad design or engineering within FESF. They aren't even usually the result of bad design or engineering of the product with which FESF is attempting to interoperate. Interoperability problems are simply an inherent part of creating File System Isolation Minifilters. Therefore, you should do what you can to set your customers' expectations appropriately.

Also, keep in mind that when customers report "everything was working until I installed your product", it does not necessarily mean that there's a problem in FESF. It is at least equally likely that some other product (the customer's antivirus or cloud filter or online backup program) is the component that's failing. Your support teams might consider explaining this to customers.

**Recommended Strategy:** Establish customer expectations that interoperability problems may occur. When you DO have an interoperability issue, get us the bug report as soon as you can along with all the data that you can provide.

### 15.2.2 Collect Good Data

When a problem does occur, we need good data with which to work so we can fix the problem expeditiously. We need to know the specific versions of the OS, including updates. We need to know everything that's running on the systems where the problem is being observed, and the versions of any third-party file system related products.

Also, we will always need problems described to us in terms of observed and expected behaviors *at the Windows file system level*. It will almost never be enough to say, "we have a client who can't copy a file to a share on a server where FESF on the server is set to encrypt the file." Of course, if that's all you have, we'll take it. But what we really need is an analysis of the behavior. What particular file system operations are failing? What errors are reported? A [ProcMon](#) trace can go a long way towards collecting this information, as can FESF V2 trace data (described elsewhere in this document).

We usually will also need one or more Windows crash dumps taken when the problem is being experienced. This has been, is now, and will always be the gold standard for documentation. Remember, when providing crash dumps, we need either full or kernel dumps. Please do not send us mini-dumps (also called “summary” dumps). They are not helpful to use in diagnosing the root cause of a problem.

**Recommended Strategy:** When you experience an interoperability issue, collect as much relevant data for us as you can up front. Always send us a description of the behaviors you’re seeing *at the Windows file system level*. Send us multiple crash dumps.

### 15.2.3 We Put Windows First

Given the vast array of potential products with which FESF could be used, and the widely varying quality of those products, we’ve developed some very clear priorities.

Here at OSR, we always prioritize supporting FESF interoperability with supported versions of Windows, and the applications that are provided in-box with those versions of Windows. This is now, and always will be, our number one priority.

If you have a Windows interoperability problem, be sure to get us the specific Windows versions involved. We will do our best to ensure a resolution is included in the next release of FESF.

When we receive a problem report involving interoperability with a third-party product, we do a preliminary triage, reviewing the data that’s submitted with the case. We spend a few hours trying to see if we can understand the issue and, assuming we do, we then look to see if there’s a reasonably quick fix. If we can’t understand the issue in the allotted time, or there’s not a reasonably quick fix evident, we put the problem aside and handle it on a “time available” basis.

While we’re pleased to receive problem reports about interoperability problems involving third-party products, our work to resolve these problems will always be given a lower priority than a Microsoft product that ships with a supported version of Windows. And we absolutely *do not* guarantee that we will fix every reported third-party interoperability problem.

Of course, we understand that sometimes you’ll have third party interoperability issues that must be fixed, and sometimes you’ll have such an issue that must also be fixed *quickly*. In such cases let us know. We can arrange to work with you to prioritize the problem appropriately, including even support at a critical level. Because such escalations are not part of our Maintenance and Support service, there will be added cost involved in such escalations.

**Recommended Strategy:** When you have a Windows interoperability problem, be sure to get us the specific version(s) of Windows involved. We’ll automatically prioritize the issue. When you contact us about an interoperability problem concerning a non-Microsoft component, get us as much data as possible *when the initial problem is reported*. Expect that, unless our initial triage yields a clear problem and solution, we will handle this on a “time available” basis. If interoperating with this third-party software is critical to your company, let us know. We can usually arrange to escalate an issue at an additional cost on a per-incident basis.

### 15.2.4 File a Bug on the OTHER Product

In many cases, regardless of the quality of the third-party product involved, the only way we can solve an interoperability problem with a third-party product is by working collaboratively with the third-party to solve the issue. This is what we do at Plugfest, as described previously. Between Plugfests (or when the other company doesn’t attend Plugfest) working collaboratively typically entails us arranging a conference call with the third-party’s development team. This collaboration may ultimately result in the third-party changing their product, OSR changing

the way FESF works (due to a discovered problem or to accommodate the way the third-party product works), or a combination of these things.

Before we can do this, we will need you or your customer to file a detailed problem report with the third-party product's vendor. Please pass along the case number of this problem report, and any associated case notes.

Because we've been working in the Windows file system field for such a long time, we often have contacts at third-party companies where interoperability problems arise. We've had very good luck working with vendors, world-wide, to collaboratively solve problems that are reported to us in this way.

**Recommended Strategy:** When you report an interoperability problem involving a third-party product to us, *always* file a bug on the third-party's product and get us the case number. Also get us any other supporting information that you can provide.

### 15.3 FESF Interoperability with Cloud Storage Products

Interoperability between FESF and Cloud Storage products, such as OneDrive and Dropbox, is unique enough that it warrants its own section of this document. While everything we've mentioned in the previous sections applies here, there are additional concerns that are solely relevant to Cloud Storage products that need to be discussed.

#### 15.3.1 Interoperability "Levels"

In testing and supporting Cloud Storage products, we describe FESF's ability to interoperate with a given Cloud Storage product as being one of three levels:

**Level 1 Interoperability (L1):** FESF and the Cloud Storage product can be installed on the same system without affecting system stability. That is, systems don't crash or exhibit unusual behaviors as a result of both products being installed simultaneously. However, regardless of the Solution policy implemented, it is not necessarily possible for FESF to encrypt files that are stored in directories that are serviced by the Cloud Storage product.

**Level 2 Interoperability (L2):** In addition to stability coexisting with a given Cloud Storage product, FESF can encrypt local copies of files that are stored in the cloud by that product. In products that use placeholder technologies (that is, where some subset of files stored in the cloud are represented by a "marker file" locally and are only fully recalled from the cloud when requested by the user), files will be encrypted by FESF when they are recalled from the cloud and stored on the local system.

**Level 3 Interoperability (L3):** FESF supports encrypting both local files and files that are stored in the cloud. This means that FESF encrypted files that are recalled from the cloud can be successfully decrypted and stored locally (in either encrypted or decrypted form) depending on the Solution policy.

We strive to ensure that all common Cloud Storage products can coexist on the same system as FESF, and thereby achieve L1 interoperability. It is sometimes hard to even achieve this level of interoperability, but it remains our goal.

Almost everyone would agree that it's better for FESF to be interoperable with a product at L2 than at L1. At L1, the best we can say is that the two products coexist peacefully. At L2 end-users can decide (depending on the capabilities of your Solution, of course) if they want their cloud files to be encrypted when they are stored on their local system. However, it's not always possible (barring major changes in the Cloud Storage product itself) for a Cloud Storage product that's interoperable with FESF at L1 today to be interoperable with FESF at L2 tomorrow.

Also, it's not clear that L3 interoperability is necessarily useful in all cases. When files are stored encrypted in the cloud (the defining attribute of L3 interoperability) users can't access them using browser-based tools such as Microsoft Office online. The ability to use these tools is one of the prime motivating factors for many end-users deciding to put their files in the cloud.

Further, it's not always possible for FESF to provide L3 interoperability with a given Cloud Storage product.

### 15.3.2 Implementing Policies and Strategies for FESF Support of Cloud Storage Products

Figuring out *how* to get FESF to support encrypting and decrypting files with a given Cloud Storage product isn't necessarily simple. It will typically involve analyzing the details of how the Cloud Storage product works, and then crafting a Solution policy that will result in FESF (and your Solution) doing what you want it to do in terms of encrypting or decrypting files. Determining how your Solution can best support a given Cloud Storage product is ultimately up to you, and is part of the value add that your Solution provides to your customers.

For example, the version of OneDrive that was current as of Windows 1809 recalls files from the cloud into a temporary location named "OneDriveTemp" and a subdirectory named by a GUID. When the file has been fully recalled, OneDrive renames it to the ultimate target directory. There are different policies and strategies that your Solution could potentially use if your goal is to cause locally stored files to be encrypted. One might involve causing all files written into the temporary directory to be encrypted. This will result in the file being encrypted when it is recalled. When the rename from the temporary directory to the ultimate target directory takes place, the file will remain encrypted. One could, however, also envision a Solution that utilizes a more sophisticated policy in which the file is first encrypted in the temporary directory with a unique key, and then when the file is renamed, that rename is intercepted and the file is re-encrypted using the ultimate end-user's key. This more advanced policy would probably be sufficiently challenging to implement that its complexity would outweigh its potential benefits. Still, the point is that such options are possible, and whether it's "worth it" is up to you.

Thus, ultimately it is up to you to analyze the behavior of a given Cloud Storage product and to devise the "best" policy and strategy for your Solution to support that product. And, don't forget, it's not unusual for different versions of a given Cloud Storage product to be implemented differently and for those different versions to be "in the wild" at the same time. We've seen this with OneDrive. There's nothing FESF can do to make this easier. It is simply incumbent on your Solution to be ready to handle these variances.

### 15.3.3 OSR's Role in Helping Configure FESF for Various Cloud Storage Products

OSR no longer provides detailed guidance or instructions on how to achieve interoperability between FESF and any given Cloud Storage Provider. This is in part due to the increasing complexity and increasingly rapid rate of change of cloud products. It is also because the way a given licensee's Solution interacts with any cloud product, and whether that Solution can achieve a given level of interoperability with that cloud product, is very much determined by the Solution, the deployment environment, the end user customer's requirements, and the applicable threat model. Because OSR cannot be aware of all these things, OSR can't provide generically applicable, definitive, guidance.

**We warn licensees** who have previously relied on OSR guidance regarding how to configure your Solution Policy to achieve interoperability with OneDrive, that **recent versions of OneDrive have significantly changed**. Among those changes, Microsoft has chosen to discontinue all support for disabling simultaneous document co-authoring. Following the step-by-step instructions provided by OSR with previous versions can lead to potential security vulnerabilities.

## 15.4 Cloud Storage Products We Test With

During Microsoft Plugfest, we test with any Cloud Storage products that are available and willing to test with us. We also do regular, ongoing, testing with some of the most common Cloud Storage products. For FESF V2.0, this includes:

- OneDrive
- Google Drive File Stream
- Dropbox Smart Sync
- Box Sync



We are always interested in hearing what Cloud Storage products are important in your markets. Please let us know, so we can consider adding them to our test matrix.

## 15.5 Summary and OSR's Recommendations

Building interoperable Windows File System products is complex. This is due to several factors, some of which we've enumerated in this paper, but all of which interact to further complicate the problem. The result is that interoperability problems are an inherent part of creating a Windows File System product. It's just the way the world is. It's why Microsoft spends so much money every year hosting Plugfest sessions.

In dealing with FESF interoperability issues, OSR's policies and recommendations are as follows:

- Establish customer expectations that interoperability problems may occur. When you DO have an interoperability issue, get us the bug report as soon as you can and collect all the data that we need. And remember: The last product that was installed before a bug was discovered is not necessarily the product that's at fault.
- When you experience *any* interop issue, collect as much relevant data for us as you can up front. Always send us a description of the behaviors you're seeing *at the Windows file system level*. Send us multiple crash dumps.
- When you have a Windows interoperability problem, be sure to get us specific versions. We'll automatically prioritize the issue, and you should typically expect a resolution in the next release of FESF.
- When you contact us about an interoperability problem concerning a non-Microsoft component, get us as much data as possible when the initial problem is reported. Expect that, unless our initial triage yields a clear problem and solution, we will handle this on a "time available" basis. If interoperating with this third-party software is critical to your company, let us know. We can usually arrange to escalate an issue at an additional cost on a per-incident basis.
- When you report an interoperability problem involving a third-party product to us, *always* file a bug on the third-party's product and get us the case number. Also get us any other supporting information that you can provide.

In addition to the above, when dealing with interoperability issues with Cloud Storage products, OSR's policy and recommendations are as follows:

- We test a variety of Cloud Storage products and have qualified them to work with FESF at different levels. The "level" (L1, L2, L3) indicates the capabilities of FESF with respect to files stored by the Cloud Storage product.
- We strive to ensure that all common Cloud Storage products can coexist on the same system as FESF and thereby achieve L1 interoperability. This may not even be possible for every product.
- Interoperability at any given level cannot always be achieved for every product. This is because the way Cloud Storage products are implemented varies widely.
- It may take detailed analysis of a product's behavior before you're able to determine a workable strategy and policy for your Solution.
- We are interested to hear which Cloud Storage products are important in your market. We are open to adding additional Cloud Storage products to our interoperability testing list.