OSR File Encryption Solution Framework

Custom Providers Kit Developer's Guide FESF V1.5

1 FESF Customer Providers Kit Overview

1.1 Introduction

The FESF Custom Providers Kit (Providers Kit) allows you to easily build per file encryption solutions in kernel mode. The kit does all the "heavy lifting" of integration with the necessary Windows sub-systems and allows you to just customize the parts which are of value to your case. This customization is done by implementing your own *Provider*.

A "Provider" is a subsystem which implements a particular piece of the functionality needed to build a per file encryption solution. The Providers Kit facilitates the development of three different types of Providers:

- **The Policy Provider** (referred to as DtSupport) – This Provider implements the encryption policy encompassing, for example:
    - Is a newly created file to be encrypted?
    - Does an individual access see a raw view, a decrypted view, or is the access denied?
    - How precisely is a newly created file to be encrypted and how should a recently created file be decrypted (Key Management)?

  You implement a Policy provider by replacing one or two modules in one of the drivers supplied in the Providers Kit.

- **The Encryption Provider** (DtCrypto). – This Provider is responsible for taking the opaque information provided by the Policy Provider, and a stream of data and encrypting or decrypting it suitably.

  You implement an Encryption Provider by replacing one module in one of the drivers supplied in the Providers Kit.

- **The On Disk Provider** (Ds) – The On Disk Provider is responsible for preserving the encrypted data and its associated metadata on disk. FESF assumes that there *is* an on disk file, but equally it is assumed that the On Disk Provider may change the size of the file on disk.

  You implement an On Disk Provider by implementing a series of functions in a dispatch table and registering that table with a driver. On Disk Providers have the most complicated role to fulfill and have to deal with several unexpected side-effects related to Windows file system logic.

Although each Provider is logically independent, there are also dependencies among the Providers. For example, a Policy Provider cannot successfully specify the use of an encryption method that the Encryption Provider doesn't support. Nor could a Policy Provider ask the On Disk Provider to save metadata that is larger than its maximum capacity. There are similar constraints between the Encryption and On Disk Providers which relate to the rounding constraints of some crypto mechanisms.

As part of the Providers Kit, we include at least one *Reference Implementation* for each type of Provider. These allows you to get started quickly and also serve as code examples of some of the issues that each of the provider types must manage.

1.2 Differences from Other OSR FESF Kits

The OSR File Encryption Solution Framework comprises a series of different kits, each with its own, specific, contents. This section describes the two primary types of FESF kits the OSR licenses.

The **FESF Custom Providers Kit** allows the development of kernel-mode code, in support of your own on access encryption solution. The kit includes the generic FESF kernel-mode source code (both for reference and for your potential customization) as well as one or more kernel-mode reference implementations of each Provider type that it supports. Using the FESF Custom Providers Kit, you can control everything about the implementation of your encryption product, from the policy that determines which view of a file a given user sees to how the encrypted form of a file is stored on the media.

In contrast to this, the **Base OSR FESF Kit** allows the development an on-access encryption product, with custom policy and encryption methods, with no kernel-mode coding required. The Base OSR FESF Kit includes (signed) binary-only versions of the FESF kernel-mode components. It also includes source code for all user-mode FESF components as well an extensive, working, sample encryption solution.

2 What's New in this Release

In addition to bug fixes, the V1.5 release of FESF includes a number of important enhancements:

2.1 Updated Toolchain

FESF now builds with Visual Studio 2017, specifically version 15.6.6.

2.2 Support for Windows 10 Spring Creators Update (RS4)

FESF V1.4 supports the Windows 10 Spring Creators Update (codename Redstone 4, Version 1803, Build number 17134.x). To accomplish this, significant changes were made in support of the numerous file system changes this new version of Windows introduces.

We remind FESF licensees that, since Microsoft introduced the concept of Windows as a service, specific releases of the Windows OS are no longer static. That means that the version of the *Windows 10 Spring Creators Update* (V1803) that you download today is not likely to be the same version that will be available for download in the same place next week. This is true even when the releases are labeled with the same build number to left of the decimal point. We therefore recommend that any Windows user who wants to maintain a stable and reliable system wait to install upgrades such as the Spring Creators Update. Microsoft had indicated that they will not push the Spring Creators Update to retail end-user systems until they judge the release is reasonably stable. We counsel corporate customers to likewise be conservative in pushing Windows upgrades to end-user desktops.

3 Notes on Implementing an On Disk Provider for UDF

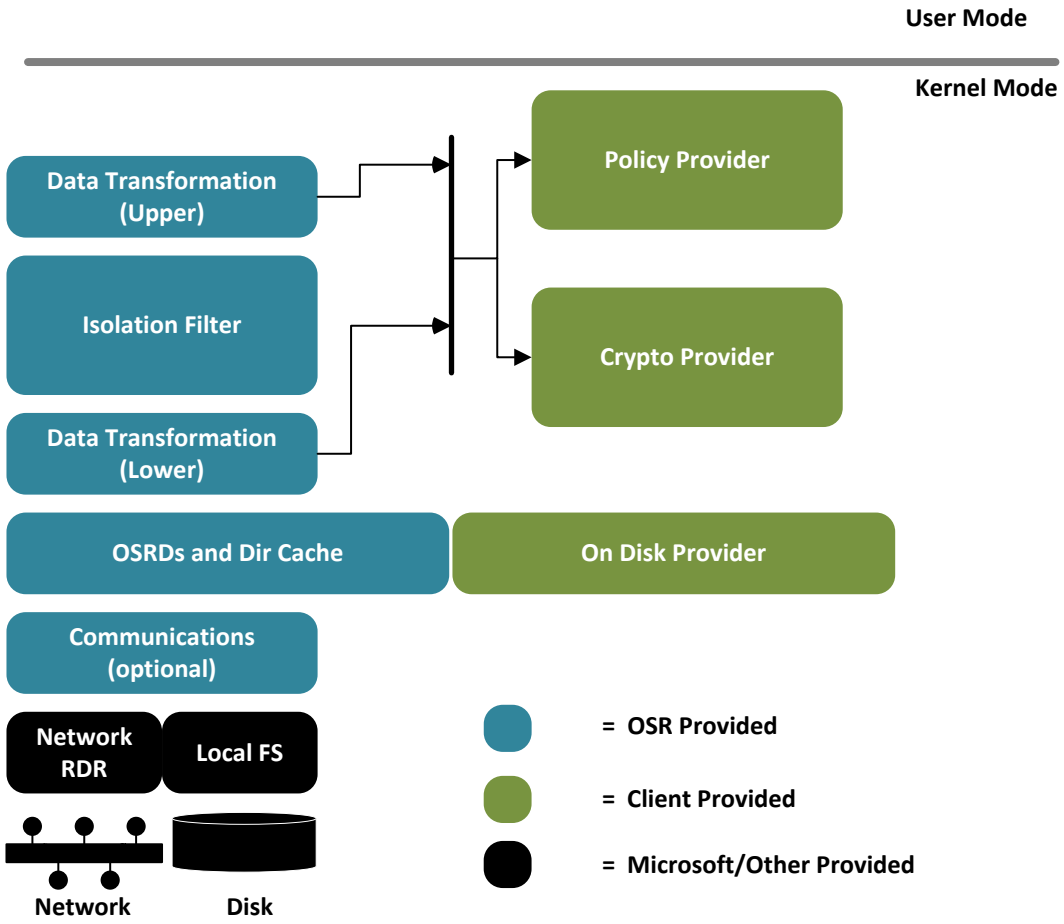
During qualification on UDF we encountered several issues around the use of UDF on removable media. In several cases we have been able to make FESF-on-top-of UDF behave better than UDF alone, particularly when it is subjected to overwriting. However great caution is advised when implementing and testing your On Disk Provider on UDF and removable media.

The greatest single issue appears to be around the handling of files when they transition from being embedded (in the UDF on disk structure) to being exbedded. Under these conditions it appears that UDF starts to cache data with no respect for coherency from the upper layers. This exacerbates its already problematic behavior under rewrite loads.

The only solution we found was to force our on disk provider to never create a file which is so small as to be embedded. This is possible since FESF allows the on disk size of the file to be hidden from the applications. We also imposed a minimum one page sector size on all UDF files implemented using our On Disk Provider.

We introduced a similar constraint into the raw path - any set eof below a certain threshold is converted first into a set eof at that threshold and then of the required size. It should be noted that as of FESF V1.3 extending writes are no longer issued by the Isolation filter which means the FileSetEndOfFileInformation is the only call to extend a file, although FileSetAllocationInformation can still truncate a file.

4 FESF Providers Kit: Architecture Overview



The FESF Custom Providers Kit consists of a stack of four (4) file system MiniFilters, all set at "altitudes" which are distinguished only by the decimal point. This means that one registered altitude can be used for all four drivers.

Each of the four MiniFilters serves a specific purpose, which is described in the following sections. Note that the default names of the MiniFilters can all be changed as part of developing your overall product.

The following sections describe each of the provided MiniFilters, starting "at the bottom" architecturally (closest to the File System).

4.1 The Support Driver

The name of this driver in the distributed kit is *OsrSupport*. It attaches at altitude *BaseAltitude+0.01*.

Within the Providers Kit, the main function of this driver is to provide functionality that needs to be shared between the other Providers Kit drivers. This is mostly related to centralized locking as well as process and thread management. As a result, this driver is implemented as a kernel-mode DLL.

Note that in the Base OSR FESF Kit (that is, *not* the Providers Kit) this driver is also used to perform various file name resolution functions which are not available in user mode.

4.2 The Data Storage (DS) Driver: On Disk Provider

The name of this driver in the distributed kit is *OsrDs2*. It attaches at altitude *BaseAltitude+0.2.*

The function of this driver is to manage any On Disk Structure (ODS) manipulation, including the separation of raw access (which the On Disk Provider never sees) from encrypted access.

The Data Storage Driver is the host for the On Disk Provider. This driver also hosts caching of directory correction entries.

It is unlikely that you will need to change any of the existing code in this driver. If you need to add an On Disk Provider, just add your code and include it according to the documentation provided later in this document. If you discover a need to change code in this driver, please contact OSR.

4.3 The Isolation Driver

The name of this driver in the distributed kit is *OsrIsolate*. It attaches at altitude *BaseAltitude+0.5*

This driver is by far the largest and most complex of the four. It contains, among other functions:

- All the logic required to interact with the Windows Cache Manager
- All the logic to ensure coherent views between encrypted and unencrypted data
- Code to finesse the issues around Transactions

It does not host any providers and does not have any configuration possible (its functioning is entirely managed from the Policy and Encryption (DT) driver).

There should never be any need to change any of the code in this driver. If you discover a need to change code in this driver, please contact OSR.

4.4 The Policy and Encryption (DT) Driver: Policy and Encryption Providers

The abbreviation “DT” stands for “Data Transformation”, which was the name by which these providers were jointly, previously known. The name of this driver in the distributed kit is *OsrDt2*. It attaches at altitudes *BaseAltitude+0.3* (encryption) and *BaseAltitude+0.8* (policy).

The primary functions of this driver are:

- To act as a host for a Policy Provider
- To act as a host for an Encryption Provider
- To interface with, and to control (based on the Policy Provider’s guidance), the functions of the Isolation Driver and the Data Storage Driver.

To add a Policy Provider, edit DtSupport.cpp and DtSupportCommon.cpp and add an Encryption Provider by editing DtCrypto.cpp. If you discover a need to change code outside these three modules, please contact OSR.

5 Reference Implementation Overview

As previously mentioned, the Providers Kit includes at least one reference implementation for each Provider type. These reference implementations are intended to illustrate the use of the interfaces provided by FESF, and to serve as a starting point for the development of your own Provider.

The reference implementations supplied for each Provider type are described in the following sections.

5.1 Policy Provider Reference Implementations

There are two Policy Provider reference implementations included as part of the Providers Kit. The first is the Providers Kit Policy Provider. The second is the kernel-mode portion of the Base OSR FESF Kit Policy Provider. These are described below.

5.1.1 The Providers Kit Policy Provider Reference Implementation

The Providers Kit Policy Provider reference implementation is contained in the files **DtSupportProvider.cpp** and **DtSupportCommon.cpp**. It implements the simplest possible, useable, policy purely for the purposes of demonstration. That policy is:

- Every new file that is created is encrypted
- Everyone gets access to encrypted files
- No operation is ever vetoed.

A registry setting provides a list of DOS devices to which the above policy will apply. The key used to encrypt a file is stored in text form as part of the metadata associated with the file.

5.1.2 The Base OSR FESF Kit Policy Provider Reference Implementation

The Providers Kit also includes the kernel-mode portion of the Policy Provider that is shipped as part of the OSR FESF Product. This code is contained in the files **DtSupport.cpp**, **DtSupportCommon.cpp** and in several additional support modules. This reference implementation is significantly more complex than the Providers Kit Policy Provider.

Because policy is determined in the OSR FESF Product in user-mode, the kernel-mode portions of the Policy Provider implement an inverted call down to user mode. While the Base OSR FESF Kit Policy Provider is included for completeness, OSR does not recommend it as a good sample from which to learn FESF or from which to start your development of your own Policy Provider. This is because it is at once not demonstrative enough and too complex to be very useful.

5.2 Encryption Provider Reference Implementation

This is the Encryption Provider used by the Base OSR FESF Kit. It uses the Windows CNG subsystem to perform encryption operations. The Encryption Provider reference implementation is configured entirely from the registry in a format which is but a thin veneer on the named property based interface natively used to configure CNG. The name which the Policy Provider communicates to the Encryption Provider is the name of a registry key in a known location which contains the parameterization for that particular provider.

The reference implementation uses ESSIV to generate the initialization vector.

The code for the Encryption Provider is contained in the file **DtCrypto.cpp**.

5.3 On Disk Provider Reference Implementation

The On Disk Provider reference implementation is contained in the file **DsSodsDispatch.cpp** and several associated modules.

This is the provider used by the Base OSR FESF Kit. Despite being the “Slim/Smart/Simple” On Disk Structure (SODS) it is complex, running to nearly 40K lines of code. This is not the place to describe how it works; rather we shall briefly describe the format it implements. For more details see the document “Simple ODS Design”.

The general idea is that the user data is prefixed with a header region which contains the metadata that the DT and crypto layers need. Additionally, we ensure that once a file has been converted to SODS format, its on-disk length will always correspond to a specific tag (that the on-disk length in hexadecimal representation always ends 0xF5).

In this implementation, application data (and metadata) is usually stored on block boundaries, but as an exception, if the data and metadata can be stored in less than 0xFF5 then the header, data and metadata are all rolled up into one appropriately sized block.

6 Getting Started

The FESF Custom Providers Kit is described by a single Microsoft Visual Studio (VS) Solution. This VS Solution builds the four drivers that comprise the kernel mode part of the FESF Solution. As shipped, it builds the Base OSR FESF Kit Policy Provider. We ship you this VS Solution to demonstrate how a real solution (the Base OSR FESF Kit) is built.

Getting started with the FESF Custom Providers Kit consists of:

- Configuring the DT project to build with the Providers Kit Policy Provider
- Building the solution
- Installing the drivers on your target machine.
- Configuring the Policy Provider and Encryption Provider appropriately

6.1 Configuring the DT project and building the solution

This step comprises removing the modules associated with the Base OSR FESF Kit Provider Policy, adding the module associated with the Providers Kit Policy Provider, and adjusting the build properties appropriately.

- Open the FESF solution in Visual Studio (with the latest WDK)
- Open the OsrDt2 project and remove the following files (which are now redundant from the project)
    - DtAccessCache.h
    - DtCommsApi.h
    - DtCommsRequests.h
    - DtCommsStructs.h
    - DtAccessCache.cpp
    - DtComms.cpp
    - DtCommsDevice.cpp
    - DtCommsRequest.cpp
    - DtCsq.cpp
    - DtSupport.cpp
- Add the following file
    - DtSupportProvider.cpp
- Select the OsrDt2 Properties and under “C/C++” select “Preprocessor”.
    - Select All configurations, All Platforms and add the following Preprocessor Definition: “NO_UMODE_COMMS”.
- Build the entire solution for your preferred Platform.

The Solution should build without warnings or errors, with /W4 and with Code Analysis (all rules) enabled in the build.

6.2 Installing and Configuring Your Target Machine.

- Move the four SYS files produced by the previous build process, and their associated INF files, to your target system and install them using the INF files. Note: DO NOT REBOOT YET.
- To Configure the Policy Provider: Apply the file **src\dt\volumes.reg**. This configures the provider to only put the F: drive into policy.
- To Configure the Encryption Provider: apply the file **src\dt\aes.reg**. This declares an algorithm called “Example-AES”. Its uses AES and has the property “ChainingMode” set to be “ChainingModeCBC”

6.3 Testing

After successfully completing the steps listed in Installing and Configuring Your Target Machine, you can reboot your target machine. When it comes back up it will be encrypting all files on the F drive. To test this, create a file on F:, then either (a) detach the driver OsrDt2 from the drive, or (b) unplug and move the F: drive to another system, and inspect the file. Note that it is indeed encrypted.

7 Customizing Providers

After you've been able to successfully build and deploy the FESF Custom Providers Kit using appropriate reference implementations for each of the Provider types, you can start customizing the Provider(s) of your choice. You do this by implementing the various callback functions described in the remainder of this document.

7.1 Custom Policy Provider

As mentioned above, you develop a Policy Provider by replacing certain APIs in the OsrDt2 drivers. These functions are defined in `DtSupport.h` and are exactly those contained in the two modules DtSupportProvider.cpp and DtSupportCommon.cpp. The choice of whether a function is in one module or another is somewhat arbitrary, but in general DtSupportProvider.cpp contains functions that you will almost certainly want to replace DtSupportCommon.cpp contains functions that will probably not want to replace, or whose function has been carefully crafted during testing three release of FESF.

7.1.1 Policy Provider Structures and Functions

The file `DtStructs.h` defines several structures which are passed to the functions below. You may wish to extend these if needed. Of particular interest are the structures associated with the Filter Manager Contexts (Instance and Stream). However, you need to be aware that the DT filter has two instances for any given volume (and hence any given object will have two contexts applicable). The structures passed to the APIS listed below are all those associated with the upper instance.

All the function names are prefixed with the word DtSupport and the documentation for each one is given in the Policy Provider Functional Reference section.

7.1.2 A Note about SRV

The default DtPrePolicy code handles create requests originating in SRV. SRV is usually made raw at this stage. If you are not editing DtPrePolicy and you want to encrypt opens from SRV, you can change this default by setting the **SrvIsRaw** value in the Parameters key of the DT driver's registry settings.

At this stage, appropriate calls will have the DT_REQUEST_FROM_SRV flag set. In order for any of this to work, the HKLM\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters\enableecp must be set to a DWORD value of one (0x01). Our sample solution sets this value during installation.

7.2 Custom Encryption Provider

The API to the Encryption Provider abstracts away the details of buffer handling in the I/O path from the implementation of encryption. The Encryption Provider therefore performs exactly two tasks: Encryption and Decryption of the provided buffers.

The implementation of an Encryption Provider can be entirely self-contained – that is to say that no detail of the implementation of the encryption need be visible outside the module(s) implementing this provider. All the information that it needs to have preserved between calls is passed around using the opaque PDT_CRYPTO_ALGORITHM structure and the opaque PDT_CRYPTO_KEY structure.

The reference implementation demonstrates how this is achieved.

7.2.1 Encryption Provider Structures and Functions

The Encryption Provider provides and consumes two data structures. The layout of these structures is entirely under the control of the implementer of the package and their contents are opaque to other parts of

the system. The documentation for these structures and the Encryption Provider functions is in the Encryption Provider Functional Reference section.

7.3 Custom On Disk Provider

In contrast to a Policy Provider, an On Disk Provider is distinct from the driver and does not share data structures; this is because, at least in theory, multiple on disk structures can be supported in a single FESF instance.

You develop an On Disk Provider by writing support functions that will then be called at appropriate times by the surrounding FESF DS MiniFilter. Your On Disk Provider's initialization code inserts pointers to these functions into a dispatch table, along with your On Disk Provider name. This dispatch table is then registered with FESF.

The DS Filter which hosts your On Disk Provider is architecturally capable of supporting multiple on disk providers, however in this version of the kit only one provider may be present and registered. This registration is done by defining (as a C++ preprocessor property) the name "SINGLE_DISPATCH" to be the name of the dispatch table you provide.

7.3.1 Registration Data Structure

The registration data structure is of type `_DS_DISPATCH_TABLE` which is defined, alongside all the types it references, in the header file `DsImplApi.h`.

```
struct _DS_DISPATCH_TABLE {
    //
    // Used to keep a list of DS implementations.  Currently
    // initialized at driver entry and never touched thereafter.
    //
    LIST_ENTRY DispatchList;

    //
    // The name is used by the calling code to differentiate
    // between drivers when we are called to stamp an ODS into
    // a stream.
    //
    UNICODE_STRING Name;

    //
    // Other function pointers are as described above.
    //
    DS_SETUP        Setup;

    DS_TEARDOWN     TeardownDs;

    DS_SETUP_VOLUME SetupVolume;

    DS_DETECT_STATE DetectState;

    DS_RECOVER_FILE RecoverFile;

    DS_READ_DT_HEADERS GetHeaders;
```

```
    DS_TEARDOWN_OBJECT TeardownStreamObject;

    DS_TEARDOWN_OBJECT TeardownVolumeObject;

    DS_CONVERT Convert;

    DS_CONVERT_EXISTING ConvertExisting;

    DS_MARK_INVALID MarkInvalid;

    DS_BOOLEAN_CALL IsValid;

    DS_SIMPLE_CALL IncrementVolatile;

    DS_SIMPLE_CALL DecrementVolatile;

    DS_SIMPLE_CALL IncrementNoWriteShare;

    DS_SIMPLE_CALL DecrementNoWriteShare;

    DS_GET_ATTRIBUTES GetAttributes;

    DS_PRE_CALL PreSetInfo;

    DS_POST_CALL PostSetInfo;

    DS_PRE_CALL PreReadWrite;

    DS_POST_CALL PostReadWrite;

    DS_CLOSE LastWriteableCleanup;

    DS_CLOSE LastWriteableClose;

    DS_IS_POTENTIALLY_ENCRYPTED IsPotentiallyEncrypted;

    DS_WRITE_DT_HEADER WriteHeader;
};
```

7.3.2 On Disk Provider Functions

In the On Disk Provider Functional Reference a typical declaration is given, rather than the typedef line which defines the types in the registration data structure.

7.3.3 Validity Management APIs

An ODS implementation will often need to know whether the on disk representation has changed or not since it was last called; this will allow it to cache on-disk information. Things which may cause the on disk representation to change include:

- A destructive create
- A raw write to the file
- At any time on the network the file may change unless all opens on this node disallow write sharing

The DS Filter and the On Disk Provider cooperate in maintaining this information. In practice, your On Disk Provider should probably just copy the implementation in the reference provider, but you are free to change it. The APIs involved are:

```
BOOLEAN
IsValid(
    _In_ PVOID StreamContext
)
```

Do we currently believe this file to be valid?

```
VOID
MarkInvalid(
    _In_ _Notnull_ PVOID VolumeContext,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ _Notnull_ PVOID StreamContext
    );
```

Something has happened to mark the file invalid.

```
VOID
DsSodsIncrementVolatile(
    _In_ _Notnull_ PVOID StreamContext
);
VOID
DsSodsDecrementVolatile(
    _In_ _Notnull_ PVOID StreamContext
);
```

The number of reasons why a Context may become invalid has gone up/down. This is usually associated with a write or a set information on a raw open.

```
VOID
DsSodsIncrementNoWriteShare(
    _In_ _Notnull_ PVOID StreamContext
);

VOID
DsSodsDecrementNoWriteShareCount(
    _In_ _Notnull_ PVOID StreamContext
);
```

The number of opens for no write share has gone up/down.

7.3.4 On Disk Support Functions

In addition to the On Disk Provider functions, there are a number of On Disk Support functions.

The Directory Correction support functions are defined/contained in:

Header: `FESFDirCorrection.h`

Lib: `Misc.lib`

The Process and Thread support functions are defined/contained in:

Header: `PsSup.h`

Lib: `OsrSupport.lib`

The functions are described in the On Disk Support Functional Reference.

8 Custom Configuration

8.1 General Customization

The INF files shipped with the Providers Kit serve as examples of how to install the four drivers which comprise the kit. There are a number of registry settings that need to be set to customize your solution and to control how the kit works. In addition, the header file `FESFConfig.h` contains some information which you may wish to change, notably:

- `MONADNOCK_PRODUCT_NAME`
- `MONADNOCK_COMPANYNAME_STR`
- `MONADNOCK_LEGALCOPYRIGHT`
- `MONADNOCK_SUPPORT_CONTACT`
- `PRODUCT_PRODUCT_RELEASE_LEVEL_STR`
- `PRODUCT_PRODUCT_RELEASE_LEVEL_STR`
- `MONADNOCK_BUILD_ENV`

These are all used to fill values in the resource files for the drivers (which in turn become user visible properties in the drivers).

Additionally, `MONADNOCK_MIN_BLOCK_SIZE` defines the unit of encryption. Since this may affect how chaining operates, you are advised to ***NEVER*** change this value after your first customer ship.

8.2 Registry Setting for SRV

`HKLM\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters\enableecp` must be set to a DWORD value of one (0x01) in order that SRV detection functions correctly.

8.3 Configuring the DT Driver

The DT Driver allows for some behavior changes to be made by changing the registry.

1. If you change the names of the drivers you need to make configuration changes to the registry for the DT Driver as described below.
2. The reference implementation for the Encryption Provider uses the `HKLM\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Algorithms` key to control its parameterization.
3. You are at liberty to use the registry to control the behavior of your providers. The Registry Path is provided to the initialization functions for both the Encryption and Policy Provider.

Other behavior of the DT Driver can be adjusted by setting values in the `HKLM\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Parameters` key.

8.3.1 Changing the Driver Names

If you decide to change the names of your drivers or instances, then the DT Driver, which choreographs the loading (if needed) and attaching of the various filters, needs to be told about this.

To change driver names, you need to provide the DT driver with the name of the Isolate and the DS drivers (defaults are OsrIsolate and OsrDs2, respectively) by setting the following REG_SZ values to the driver names:

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

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

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

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

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

8.3.2 Registry Parameters to Customize the DT Driver

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Parameters

Value Name: ByPassProcesses
Value Type: REG_MULTI_SZ
Value: <Lists processes to bypass >

This REG_MULTI_SZ registry value is used to provide input to the function:

```
BOOLEAN
DtIsExcludeProcess(
    PUNICODE_STRING Name
    );
```

The reference implementation makes use of this in the `DtSupportPrePolicy` callback (in `DtSupportCommon.cpp`).

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Parameters

Value Name: SrvIsRaw
Value Type: DWORD
Value: <Controls type of access given to encrypted files by client>

This REG_DWORD registry value is used to provide input to the function.

```
BOOLEAN
DtIsSrvRaw(
    VOID
    );
```

The reference implementation makes use of this in the `DtSupportPrePolicy` callback (in `DtSupportCommon.cpp`).

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<DtDriverName>\Parameters

Value Name: OnDiskyStructure
Value Type: REG_SZ
Value: <Lists processes to bypass >

This REG_SZ registry value is used to provide input to the function.

```
UNICODE_STRING
DtDsOds(
    VOID
    );
```

This value is then passed down to the DS driver and used to select the On Disk Provider. Hence the name here has to match the `Name` field in the `DS_DISPATCH_TABLE` structure.

8.3.3 Refreshing the Parameters

Calling the function:

```
NTSTATUS
DtReloadRegistry(
    VOID
    )
```

Will cause the registry to be re-read and the parameterization to be reset.

8.4 Configuring the DS Driver

Only one small part of the DS behavior (apart from anything you may seek to add to your provider) can by controlled by configuring the registry: That is the maximum size of the cache for any one directory. This can be controlled by the value,

`HKLM\SYSTEM\CurrentControlSet\Services\<DsDriverName>\DirCacheMaxSize`

with 0 meaning “suppress the caching” and non-zero values defining the size of the cache that each cached directory will have. “Size” means the number of corrected entries that will be stored (we do not cache the information for unencrypted files, that is for the FSD to do). Each entry consumes about 64 bytes of paged memory. The default value is 300.

Coding and API Reference

Policy Provider Functional Reference

DtInitializeDtSupport and DtTeardownDtSupport

```
NTSTATUS
DtInitializeDtSupport(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
    );

VOID
DtTeardownDtSupport(
     VOID
     );
```

Parameters

The parameters to `DtInitializeDtSupport` are those supplied during driver initialization.

Remarks

These functions are called during driver initialization and driver teardown, respectively. As an aside, developers are encouraged to frequently unload the DT during development. So long as the DT is run with Verifier this provides an excellent test against resource leakage.

Unloading the DT will cause the rest of the FESF System (Isolate and DS) to quiesce.

DtSupportAttach

```
NTSTATUS
DtSupportAttach(
    _In_ PFLT_VOLUME Volume,
    _In_ PFLT_INSTANCE Instance,
    _Out_ BOOLEAN *MustSetToFalse
    );
```

Parameters

The first two parameters are taken from the PCFLT_RELATED_OBJECTS FltObjects parameter to the Filter Manager Instance setup call.

The first parameter **must** be set to FALSE.

Return Value

STATUS_SUCCESS if the volume will contain encrypted files.
STATUS_FLT_DO_NOT_ATTACH if the volume will not contain encrypted files.

Remarks

The implementation of this function should inspect the PFLT_VOLUME provided by the parameter and determine whether FESF will be supporting encrypted files on the volume. Support is indicated by the return parameter.

DtSupportDetach

```
VOID
DtSupportDetach(
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _In_ FLT_INSTANCE_TEARDOWN_FLAGS Flags,
    _In_ _NotNull_ PVCB Vcb
    );
```

Parameters

The first two parameters are taken from the parameters to the `InstanceTeardownCompleteCallback` callback,

The third parameter is the (upper) DT's VCB

Return Value

VOID function.

Remarks

This function is called after Filter Manager and FESF have completed detach processing.

DtSupportPrePolicy

```
MONADNOCK_DISPOSITION
DtSupportPrePolicy(
     _In_ PDT_VCB Vcb,
     _In_ PFLT_CALLBACK_DATA CallbackData,
     _In_ PCFLT_RELATED_OBJECTS FltObjects
    );
```

Parameters

Vcb

Contains per volume information (it is the Filter Manager instance context).

CallbackData

FltObjects

Come directly from the parameters passed to the Filter Manager pre-create call.

Return Value

MONADNOCK_DISPOSITION

Remarks

This function gives the Policy Provider the option to shortcut processing by deciding early on that a file/stream need not be processed. It is called in pre-create in the caller's thread context. The implementation of this function should inspect the provided parameter and return the appropriate MONADNOCK _DISPOSITION.

The enum MONADNOCK _DISPOSITION is defined as

```
 enum class MONADNOCK_DISPOSITION : USHORT {

    BypassUnsafe = 54,    // No caching (Isolation), no Ds
    Raw,                  // Cached (Isolation), NoEncryption(Dt), No Ds work.
    Normal = 57,          // In: No preference (in).
                          // Out: Cached (Isolation), Encrypted(Dt - potential null), Ds
work.
       PerFileCorrect        // Internal Disposition for Dt handling.
};
```

If an implementation returns anything but **DispositionNormal**, none of the following functions will be called.

In practice all implementation should return **MONADNOCK_DISPOSITION::Normal** or, **MONADNOCK_DISPOSITION::Raw**; **MONADNOCK_DISPOSITION::BypassUnsafe**should be avoided since it completely bypasses the caching and cache coherency provided by the rest of FESF. Further, renaming of bypass files is very problematic.

It should be noted that the reference implementation of this is in two stage. First a call is made to `DtSupportProviderPrePolicy, and then a series of checks is made against paths which our testing has shown to be best made bypass.`
`It is recommended that you preserve this format, implementing your own version of DtSupportProviderPrePolicy only. It is likely that the paths which are known to cause issues will change with each following release as issues are worked around and other issues discovered and keeping this separation will allow you to leverage our testing.`

DtSupportProbeCreate

```
BOOLEAN
DtSupportProbeCreate(
    _In_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _In_ PDT_VCB Vcb
    );
```

Parameters

Data

Describes the create which is about to be performed.

FltObjects

Contains miscellaneous Filter Manager structures relating to this request

Vcb

Contains per volume information (it is the Filter Manager instance context)

Return Value

TRUE to provoke being called back at **DtSupportVetoProbedCreate**, FALSE otherwise

Remarks

This function works in tandem with **DtSupportVetoProbedCreate** to give the DT the opportunity to deny an open on a file based on information which requires an open file object to the file before the requested operation proceeds which operation might result in the file being destroyed, for instance by the SUPERSEDE or DELETE_ON_CLOSE dispositions.

It is thus called during pre-create processing.

Returning TRUE from this function causes the DT to be called at **DtSupportVetoProbedCreate** with a usable **PFILE_OBJECT** and a **HANDLE**.

DtSupportVetoProbedCreate

```
NTSTATUS
DtSupportVetoProbedCreate(
    _In_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _In_ PDT_CREATE_CONTEXT Context
    );
```

Parameters

Data

Describes the create which is about to be performed.

FltObjects

Contains miscellaneous Filter Manager structures relating to this request

Context

Contains some fields which may be useful in influence. Specifically:

Vcb

Contains per volume information (it is the filter manager instance context)

Handle

A HANDLE opened to the file (for no access and full share). The DS layer has seen this as a RAW open.

FileObject

A pointer to a FILE_OBJECT associated with Handle

All other fields in the Context structure should not be consulted

Return

If the function returns an error status then the create described by Data is failed with this status. Only an error status or STATUS_SUCCESS can be returned.

This function is called during pre-create processing if a preceding call to **DtSupportProbeCreate** returned TRUE.

If this function returns a failure status then user open request will be failed with that status before any data has been altered.

Restrictions

The File Object can only be used to perform a limited set of functions.

- Because the file object is 'raw', no I/O will be passed to a custom DS.
- Because the file object is for a no-access open, I/O cannot be performed over the network. In this case an open of "" relative to the handle should be performed. Such a create will be seen by the lower layers as a Bypass (a type of raw) create. Bear in mind that this create might also provoke a sharing violation which would otherwise not have happened unless the DesiredAccess and ShareAccess are minimized to that in the Data parameter.
- If using the lower file object for I/O only uncached I/O can be performed., cached I/O or use of sections (ZwCreateSection) will deadlock the system.
- Paging I/O should not be performed.

DtSupportCreateStream

```
NTSTATUS
DtSupportCreateStream(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_ _Notnull_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ _Notnull_ PFLT_FILE_NAME_INFORMATION FileName,
    _In_ ULONG Flags,
    _In_ ACCESS_MASK GrantedAccess,
    _In_ ULONG CreateAction,
    _Out_ _Notnull_ PDT_CRYPTO_ALGORITHM *Algorithm,
    _Out_ PVOID *KeyMaterial,
    _Out_ ULONG *KeyMaterialSize,
    _Out_ PVOID *DtHeader,
    _Out_ ULONG *DtHeaderSize
    );
```

Input Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is the Filter Manager instance context)

Scb

Contains per stream information (it is the Filter Manager stream context)

FileObject

The file object associated with the request. When this call is made FileObject represents an open handle to the file.

FileName

The name of the file. It will have been normalized unless the normalization could not be done in which case the Flags parameter will have the DT_NAME_NOT_NORMALIZED bit set

Flags

The bitwise or of three flags:

DT_NAME_NOT_NORMALIZED As noted above, the FileName parameter is usually the normalized name. If the normalization failed then this flag is set.

DT_REQUEST_FROM_SRV This flag is set if the request originated from Srv. HKLM\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters\enableecp must be set to a DWORD value of one (0x01). Our sample solution sets this value during installation

DT_EFS_FILE This flag is set if the file has been determined to be encrypted by EFS.

GrantedAccess

The access mode that has been granted to the file

CreateAction

The created disposition of the file (FILE_CREATED, FILE_OVERWRITTEN and so forth).

Output Parameters

Algorithm

A pointer to an opaque data structure which describes how the file is to be encrypted, this is described in the reference section for PDT_CRYPTO_ALGORITHM.

KeyMaterial
KeyMaterialSize

Define a buffer which contains the key material to be used to encrypt the file.

DtHeader
DtHeaderSize

Define a buffer which is written in cleartext to the stream. The DtSupport module is expected to be able to generate KeyMaterial from this header on subsequent calls to DtSupportLookupStream.

Remarks

This function is called in post create, when a zero length file has been encountered and for which no registered DS implementation has recognized. The implementation of this function has to decide whether the file is going to be encrypted (when it is first written to) and if so, what key material should be used and what header should be stored.

If the file/stream is not to be encrypted then the function should return DT_IGNORE, otherwise it should return STATUS_SUCCESS and suitably populate the output parameters. Other failures will be returned to the application calling create.

Note Regarding Synchronization

The FESF system provides no guarantees that this callback may not be active for the same file or stream across multiple threads and machines. The Data Storage layer is expected to provide the synchronization such that the "first writer wins". Thus, there is no guarantee that the returned header and key material will be used, instead the header and key material from another call may be used, or if the header was written on a remote node, the key material returned by a call to DtSupportCheckDirBehavior.

DtSupportCheckAccess

```
NTSTATUS
DtSupportCheckAccess(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_ _Notnull_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ _Notnull_ PFLT_FILE_NAME_INFORMATION FileName,
    _In_ ULONG Flags,
    _In_ ACCESS_MASK GrantedAccess,
    _In_ ULONG CreateAction,
    _In_ PVOID DtHeader,
    _In_ ULONG DtHeaderSize
    );
```

Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is a Filter Manager instance context).

Scb

Contains per stream information (it is a Filter Manager stream context).

FileObject

File object associated with the request. When this call is made FileObject represents an open handle to the file.

FileName

Name of the file. It will have been normalized unless the normalization could not be done in which case the Flags parameter will have the DT_NAME_NOT_NORMALIZED bit set.

Flags

As described for DtSupportCreateStream.

GrantedAccess

The access mode that has been granted to the file.

CreateAction

The created disposition of the file (FILE_CREATED, FILE_OVERWRITTEN and so forth).

DtHeader
DtHeaderSize

Defines a buffer whose contents were supplied as output from a previous call to DtSupportCreateStream

Remarks

This function is called in post-create for encrypted files. The implementation of this function may choose to deny access to the caller (by return STATUS_ACCESS_DENIED) or allow access to the encrypted or decrypted data in the file (by return STATUS_SUCCESS) or to only allow unencrypted access (by returning DT_RAW).

This function may be called after a call to DtSupportCreateStream (if relevant) and will be called before a call to DtSupportLookupStream (if relevant).

DtSupportCheckDirBehavior

```
NTSTATUS
DtSupportCheckDirBehavior(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_ _Notnull_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ _Notnull_ PFLT_FILE_NAME_INFORMATION FileName,
    _In_ ULONG Flags,
    _In_ ACCESS_MASK GrantedAccess,
    _In_ ULONG CreateAction
    );
```

Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is a Filter Manager instance context)

Scb

Contains per directory information (it is a Filter Manager stream context)

FileObject

The file object associated with the request. When this call is made, FileObject represents an open handle to the directory.

FileName

The name of the directory. It will have been normalized unless the normalization could not be done in which case the Flags parameter will have the DT_NAME_NOT_NORMALIZED bit set.

Flags

As described for DtSupportCreateStream.

GrantedAccess

The access mode that has been granted to the directory

CreateAction

The created disposition of the file (FILE_CREATED, FILE_OPENED and so forth).

Remarks

This function is called in post-create for directories.

The implementation of this function may choose to deny access to the directory (by returning STATUS_ACCESS_DENIED) or to allow the directory to be listed showing the real on disk sizes (by returning DT_RAW). It can also arrange that all entries are unconditionally corrected (by returning STATUS_SUCCESS) or for the function DtSupportPerFileCorrect to be called for each directory listing (by returning DT_PER_FILE_CORRECT).

DtSupportPerFileCorrect

```
BOOLEAN
DtSupportPerFileCorrect(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_opt_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT DirectoryFileObject,
    _In_bytecount_(DirectoryBufferSize) _Notnull_ PVOID DirectoryBuffer,
    _In_ ULONG DirectoryBufferSize,
    _In_ FILE_INFORMATION_CLASS InfoClass,
    _In_ KPROCESSOR_MODE RequestorMode
    );
```

Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is the Filter Manager instance context).

Scb

Contains per stream information (it is the Filter Manager stream context).

DirectoryFileObject

The file object associated with the request.

DirectoryBuffer
DirectoryBuffersize

Describe the directory buffer under consideration. This is one of the standard directory enumeration formats as described by InfoClass

InfoClass

The type of (directory) enumeration.

RequestorMode

The mode (kernel or user) of the directory enumeration request. This allows the buffer to be probed correctly, which operation (and all buffer manipulation) should be done within a try/catch statement

NOTE : The DirectoryBuffer, DirectoryBufferSize, and InfoClass parameters are suitable for being passed iteratively to FesfGetNextDirectoryEntry.

Return Value

This function should return FALSE only if the output requires no further processing (either because no entry was found to be flagged or because all flagged entries had that flag cleared).

The example code in DtSupportProvider.cpp shows how to iterate across a buffer and unconditionally require that every entry be corrected.

Remarks

This function is called during directory enumeration for those directories where the call to DtSupportCheckDirBehavior returned DT_PER_FILE_CORRECT.

This function is responsible for traversing the provided buffer (in a try-catch-finally statement), probably using the FesfGetNextDirectoryEntry function and calling FesfClearForCorrection for those entries which do not require size correction. Whether a file is a candidate for correction can be checked with the FesfIsMarkedForCorrection function.

DtSupportLookupStream

```
NTSTATUS
DtSupportLookupStream(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_ _Notnull_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ _Notnull_ PFLT_FILE_NAME_INFORMATION FileName,
    _In_ ULONG Flags,
    _In_ PVOID DtHeader,
    _In_ ULONG DtHeaderSize,
    _Out_ _Notnull_ PDT_CRYPTO_ALGORITHM *Algorithm,
    _Out_ PVOID *KeyMaterial,
    _Out_ ULONG *KeyMaterialSize
    );
```

Input Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is the Filter Manager instance context).

Scb

Contains per stream information (it is the Filter Manager stream context).

FileObject

The file object associated with the request. When this call is made, FileObject represents an open handle to the file.

FileName

The name of the file. It will have been normalized unless the normalization could not be done in which case the Flags parameter will have the DT_NAME_NOT_NORMALIZED bit set.

Flags

As described for DtSupportCreateStream.

DtHeader

DtHeaderSize

Defines a buffer whose contents were supplied as output from a previous call to DtSupportCreateStream

Output Parameters

Algorithm

A pointer to an opaque data structure which describes how the file is to be encrypted. This is described more in the section about CryptoProviders.

KeyMaterial
KeyMaterialSize

Define a buffer which contains the key material to be used to encrypt the file. This is described more in the section about CryptoProviders.

Remarks

This function is called when the FESF system encounters a file/stream which has been previously converted via (successful) call to DtSupportCreateStream. The implementation is given the header which was previously returned and has to return the associated key material and algorithm. The MiniFilter caches the material in the SCB and makes it available to the encryption callbacks.

DtSupportRename

```
NTSTATUS
DtSupportRename(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_opt_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ PFLT_FILE_NAME_INFORMATION FileName,
    _In_ PFLT_FILE_NAME_INFORMATION NewFileName,
    _In_ ULONG Flags,
    _Out_ _Notnull_ BOOLEAN PostCall
    );
```

Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is the Filter Manager instance context).

Scb

Contains per stream information (it is the Filter Manager stream context).

FileObject

The file object associated with the request. When this call is made, FileObject represents an open handle to the file.

FileName

The name of the file. It will have been normalized unless the normalization could not be done in which case the Flags parameter will have the DT_NAME_NOT_NORMALIZED bit set.

NewFileName

The normalized name that will result from a successful rename.
Not that if the DT_TARGET_NAME_INVALID bit is set in Flags, only the Volume and Share fields are valid.

Flags

As described for DtSupportCreateStream.
Additionally DT_TARGET_NAME_INVALID is set if Filter Manager could not determine the target. This is often associated with a rename "through" a symbolic link to a share

PostCall

If set to TRUE then the DtSupportPostRename API will be called after the rename has happened.

Remarks

This function is called in the pre- `IRP_MJ_SET_INFORMATION` path when any arbitrary rename is encountered. The purpose of this call is to give the Policy Provider the opportunity to veto the rename.

DtSupportPostRename

```
NTSTATUS
DtSupportRename(
    _In_ NTSTATUS Status,
    _In_opt_ PDT_SCB Scb,
    _In_ FLT_POST_OPERATION_FLAGS Flags
    );
```

Parameters

Status

The result of the operation.

Scb

Contains per stream information (it is the Filter Manager stream context).

Flags

The flags passed in to the operation by the FilterManager.

Remarks

This function is called in the post-rename path if *PostRename was set to TRUE in a call toDtSupportRename. It is called in the same context as the pre-rename but without the “TLS” being set

DtSupportLinkCreate

```
NTSTATUS
DtSupportLinkCreate(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_opt_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ PFLT_FILE_NAME_INFORMATION FileName,
    _In_ PFLT_FILE_NAME_INFORMATION LinkFileName,
    _In_ ULONG Flags
    );
```

Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is the Filter Manager instance context).

Scb

Contains per stream information (it is the Filter Manager stream context).

FileObject

The file object associated with the request. When this call is made, FileObject represents an open handle to the file.

FileName

The name of the file. It will have been normalized unless the normalization could not be done in which case the Flags parameter will have the DT_NAME_NOT_NORMALIZED bit set

LinkFileName

The normalized name that will result from a successful link create
Not that if the DT_TARGET_NAME_INVALID bit is set in Flags, only the Volume and Share fields are valid.

Flags

As described for DtSupportCreateStream.
Additionally DT_TARGET_NAME_INVALID is set if Filter Manager could not determine the target. This is often associated with a rename “through” a symbolic link to a share

Remarks

This function is called in the pre-`IRP_MJ_SET_INFORMATION` path when an attempt is going to be made to create a hard link to a file. The purpose of this call is to give the Policy Provider the opportunity to veto the link create.

DtSupportLastCleanup

```
VOID
DtSupportLastCleanup(
    _In_ _Notnull_ PDT_SCB Scb,
    _In_ FLT_POST_OPERATION_FLAGS Flags     );
```

Parameters

Scb

Contains per stream information (it is a Filter Manager stream context).

Flags

The flags passed in to the operation by the FilterManager.

Remarks

This function is called in the POST CLEANUP path when the count of open handles (the number of IRP_MJ_CREATE operations unmatched by a IRP_MJ_CLEANUP) drops to zero.

We do not expect that this API will be widely used.

DtSupportCorruptFile

```
NTSTATUS
DtSupportCorruptFile(
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_ PFLT_FILE_NAME_INFORMATION FileName
    );
```

Parameters

Vcb

Contains per volume information (it is the Filter Manager instance context).

FileName

Gives details about the failing file.

Remarks

This function is called when an On Disk Provider returns the distinguished error DT_CORRUPT. This API **MUST** return either a failure status (which will be reflected to the calling application) or DT_RAW in which case the application will get unencrypted access to the file.

DtSupportTransactedOpen

```
NTSTATUS
DtSupportTransactedOpen(
    _In_ _Notnull_ PETHREAD Thread,
    _In_ _Notnull_ PDT_VCB Vcb,
    _In_ _Notnull_ PDT_SCB Scb,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ _Notnull_ PKTRANSACTION Transaction,
    _In_ _Notnull_ PFLT_FILE_NAME_INFORMATION FileName,
    _In_ ULONG Flags,
    _In_ ACCESS_MASK GrantedAccess,
    _In_ ULONG CreateAction
    );
```

Parameters

Thread

The pointer to the current thread.

Vcb

Contains per volume information (it is the Filter Manager instance context).

Scb

Contains per stream information (it is the Filter Manager stream context).

FileObject

The file object associated with the request. When this call is made, **FileObject** represents an open handle to the file.

Transaction

Describes transaction under which this create has happened.

FileName

The name of the file. It will have been normalized unless the normalization could not be done in which case the Flags parameter will have the DT_NAME_NOT_NORMALIZED bit set. You should be aware that this is the name of the file within this transaction.

Flags

As described for DtSupportCreateStream.

GrantedAccess

The access mode that has been granted to the file.

CreateAction

The created disposition of the file (FILE_CREATED, FILE_OVERWRITTEN and so forth).

Remarks

All transactional created files are ignored by the FESF subsystem (by marking them bypass), and this is applied retroactively (all existing handles to the same file are marked as bypass). This is the technology that we have discovered as being required to allow Windows Update to work.

Since this gives an unprivileged user a simple way to gain raw access to a file, this call exists to allow you to veto transacted opens, it is called during post create processing.

You should be aware that vetoing transactions is liable to make Windows Update fail.

DtSupportFsCtl

```
PFLT_PREOP_CALLBACK_STATUS
DtSupportFsCtl(
    _In_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects
    );
```

Input Parameters

Data
FltObjects
The standard parameters supplied to Filter Manager precalls.

Returns

This function must return one of:

- FLT_PREOP_SUCCESS_NO_CALLBACK: to indicate that the FsCtl has not been recognized.
- FLT_PREOP_COMPLETE: to indicate that it has recognized and processed the request and set up the Io Status Block in the PFLT_CALLBACK_DATA
- FLT_PREOP_PENDING : to indicate that it has posted the request and will complete it (using FltCompletePendedPreOperation) at a later stage

Remarks

This callback is called in the pre-call processing to allow the Provider to intercept file system controls targeted at it.

DtSupportGetEventLogDevice

```
PDEVICE_OBJECT
DtSupportGetEventLogDevice(
    VOID
    );
```

If this returns a non null value, then the driver will log errors to this device using `IoWriteErrorLogEntry`.

DtSupportDestroyKeyMaterial

```
VOID
DtSupportDestroyKeyMaterial(
    _In_ PVOID KeyMaterial,
    _In_ ULONG KeyMaterialSize
    );
```

This method is called to destroy key material returned by DtSupportLookupStream or DtSupportCreateStream. The DT may choose to scrub the memory before freeing it.

DtSupportFreeHeader

```
VOID
DtSupportFreeHeader(
    _In_ PVOID Header,
    _In_ ULONG HeaderSize
    );
```

This method is called to free up the Header returned by DtSupportCreateStream (and subsequently stored on disk). Since this material is not sensitive, the DT need not scrub the memory before freeing it.

Encryption Provider Functional Reference

PDT_CRYPTO_ALGORITHM

This structure refers to a particular encryption algorithm (“AES-128 with CBC”, or “Elliptic curve using a NIST/X9.62/SECG curve over a 192 bit prime field” and so forth).

This structure may be allocated from paged pool. It can be created in any way that an implementation wants. An example is given in `DtCryptoLookupAlgorithm.`

PDT_CRYPTO_KEY

This structure refers to a particular instance of an algorithm, differentiated by the key material supplied.

It should be allocated from non paged pool. It is reference counted; when first returned (by `DtCryptoCreateKey`) a reference of 1 is assumed. When the count goes down to 0 the structure can be freed.

The calling system does not guarantee that two increments will not happen at once, nor will it guarantee that two decrements happen at once and so interlocked instructions should be used to maintain the reference count. It will however guarantee than no increment will happen while the reference count is dropping from 1 to 0, and that once the reference count has dropped to zero it will never be incremented.

The key contains sensitive information and it is usual to scrube the buffer prior to being freed. Since the buffer is allocated from non paged pool it will never be written to the paging file.

DtCryptoInit and DtCryptoTeardown

```
NTSTATUS
DtCryptoInit(
    _In_ PUNICODE_STRING RegistryPath
    );

VOID
DtCryptoTeardown(
    VOID
    );
```

These functions are called at driver initialization and unload respectively.

DtCryptoSecondaryInitialize and DtCryptoIsSecondaryInitialized

```
NTSTATUS
DtCryptoSecondaryInitialize(
    VOID
    );

BOOLEAN
DtCryptoIsSecondaryInitialized(
    VOID
);
```

DtCryptoSecondaryInitialize allows deferred initialization. There are some CNG operations which cannot take place during driver initialization. DtCryptoSecondaryInitialize is called during create processing so long as DtCryptoIsSecondaryInitialized has returned FALSE.

The reference implementation uses this callback to initialize the ESSIV generation.

DtCryptoLookupAlgorithm

```
NTSTATUS
DtCryptoLookupAlgorithm(
    _In_ PUNICODE_STRING Name,
    _Out_ PDT_CRYPTO_ALGORITHM *Algorithm
    );
```

This is a helper function in the Encryption Provider reference Implementation which demonstrates how to create a CNG BCRYPT_ALG_HANDLE from an arbitrary registry setting.

DtCryptoCreateKey

```
NTSTATUS
DtCryptoCreateKey(
    _In_ PDT_CRYPTO_ALGORITHM Algorithm,
    _In_ PVOID KeyMaterial,
    _In_ ULONG KeyMaterialSize,
    _Out_ PDT_CRYPTO_KEY *Key
    );
```

Given the algorithm and some key material, create something which can be passed to later calls to `DtCryptoCodeMdl.`

Input Parameters

Algorithm

Some data structure which has been separately generated.

KeyMaterial

KeyMaterialSize

The raw key material to be used when encrypting and decrypting.

Output Parameters

Key

An opaque, referenced data structure that can be passed to `DtCryptoCodeMdl`.

DtCryptoRefKey

```
VOID
DtCryptoRefKey(
    _In_ PDT_CRYPTO_KEY Key
    );
```

This function increments the reference count on the supplied key.

DtCryptoDerefKey

```
VOID
DtCryptoDerefKey(
    _In_ _CRYPTO_KEY Key
    );
```

This function decrements the reference count on the supplied key. When the reference count drops to zero the structure can be deallocated.

DtCryptoCodeMdl

```
NTSTATUS
DtCryptoCodeMdl(
    _In_ PDT_SCB Scb,
    _In_ PMDL InMdl,
    _In_ PMDL OutMdl,
    _In_ ULONG Size,
    _In_ LARGE_INTEGER StartOffset,
    _In_ BOOLEAN Encrypt,
    _In_ PUCHAR WorkSpace
    );
```

Parameters

Scb

The DT Stream Context. Several fields are of interest to code implementing encryption.

Scb->CryptoKey
Refers to the CryptoKey returned by DtCryptoCreateKey.

InMdl

MDL defining the input Buffer (plaintext when **Encrypt** is TRUE).

OutMdl

MDL defining the output Buffer (plaintext when **Encrypt** is FALSE).

Size

The number of bytes to be encoded and taken from the InMDL (Encryption) or placed into the OutMDL (decryption).

StartOffset

The offset of the read or write, and can be used as an IV.

Encrypt

Controls whether the operation is encryption (TRUE) or decryption(FALSE).

WorkSpace

A buffer of size `FESF_MIN_BLOCK_SIZE` provided as workspace to allow the encryption and decryption to deal with issues to do with unaligned buffers and writes. See below.

Remarks

All Encryption and Decryption is performed via calls to this function.

In most cases encryption and decryption will be in done in the paging path and the size will be multiples of 4K. In some circumstances (for instances files small enough to fit into the MFT or non buffered I/O) size will not be aligned. In this case, the DT has to over encrypt or under decrypt in order to respect the block size of the particular algorithm.

In the ***Encryption*** path, the **OutMDL** will be large enough to contain data up to next `FESF_MIN_BLOCK_SIZE` boundary. If the size is unaligned, the encryption is expected to copy the last part of the buffer into WorkSpace, zero pad it and then encrypt the whole into the appropriate part of the **OutMDL .**

In the ***Decryption*** path, the **InMDL** will be large enough to contain data up to next `FESF_MIN_BLOCK_SIZE` boundary. If the size is unaligned, the encryption is expected to decrypt the last part of the buffer into WorkSpace, and then copy the unaligned part into the appropriate part of the **OutMDL.**

The On Disk Provider is aware of the encryption size and will store enough of the buffer beyond **Size** to allow the encryption to work – it explicitly allocated and stores enough bytes to round up to the size defined by `DtCryptoBlockSize.` For example, if an encrypt of 20 bytes comes in, **OutMDL** will have enough space to contain `FESF_MIN_BLOCK_SIZE` bytes. The encryption copies 20 bytes to the workspace, zeros the rest of the buffer and encrypts `FESF_MIN_BLOCK_SIZE` bytes appropriately into **OutMDL**. DS will store enough of the buffer to ensure that the subsequent decryption works. On decrypt, **InMDL** has enough space to contain `FESF_MIN_BLOCK_SIZE` bytes. The decryption decrypts `FESF_MIN_BLOCK_SIZE` bytes to the workspace and then copies 20 bytes into **OutMDL**.

DtCryptoBlockSize

```
ULONG
DtCryptoBlockSize(
    _In_ PDT_CRYPTO_ALGORITHM Algorithm
    );
```

This function returns the natural block size for the algorithm in question. It is used to instruct the On Disk Provider to over-allocate space for files to allow for up to that number of bytes.

The returned size should be a power of 2 and less than `FESF_MIN_BLOCK_SIZE` which is upper bound on block size.

For example an AES-256 implementation requires that the input data be encrypted in 256bit (==8 bytes) blocks, and so to encrypt a 15 byte file would require a 16 byte input and output. In this case the BlockSize is 8.

On Disk Provider Functional Reference

In the sections that follow, a typical declaration is given, rather than the typedef line which defines the types in the Registration Data Structure .

SetupDs

```
NTSTATUS
SetupDs(
    _In_ _Notnull_ PDRIVER_OBJECT DriverObject,
    _In_ _Notnull_ PUNICODE_STRING RegistryPath
    );
```

This function is used to initialize the On Disk Provider execution. It is used to allocate any global resources that might be used by the On Disk Provider component. It is called during initialization.

The parameters are those supplied to DriverEntry.

TeardownDs

```
VOID
TeardownDs(
    VOID
    );
```

This function is used to terminate the DS execution. It is used to free resources that might have been allocated by the On Disk Provider.

SetupVolume

```
NTSTATUS
SetupVolume(
   _In_ PCFLT_RELATED_OBJECTS FltObjects,
   _In_ FLT_FILESYSTEM_TYPE VolumeFilesystemType,
   _In_ BOOLEAN IsNetwork,
   _Inout_ PVOID* VolumeContext
);
```

Input Parameters

FltObjects

VolumeFileSystemType

As supplied to Instance Setup (PFLT_INSTANCE_SETUP_CALLBACK) in Filter Manager.

IsNetwork

Defines whether the volume is a network device or a local volume.

Output Parameter

VolumeContext

A totally opaque object which will then be passed (as the "VolumeContext" parameter) to other callbacks.

Remarks

The purpose of this function is to permit establishing a per-volume context structure for use by the On Disk Provider. This value is passed into subsequent calls to indicate per-volume context.

Examples of per-volume context would include:

- Type of file system and/or characteristics
- Cluster (allocation unit) size of file system
- Sector (atomic I/O unit) size of file system

TeardownVolumeObject

```
VOID
TeardownVolumeObject(
    _In_ PVOID Object
    );
```

This frees up any resources, and deallocates the context returned by `SetupVolume`.

DetectState

```
NTSTATUS
DetectState (
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _Inout_ BOOLEAN *Converted,
    _Post_ _When_(*Converted, _Out_) PVOID *DtHeader,
    _Post_ _When_(*Converted, _Out_) ULONG *DtHeaderLength,
    _Post_ _When_(*Converted, _Out_) PVOID *StreamContext
   );
```

Input Parameters

FltObjects

Passed from Filter Manager, notably giving access to the file object a Filter Manager instance.

Output Parameters

These are described below

Returns

STATUS_SUCCESS

The file was successfully processed (regardless of whether it was recognized or not).

DT_CORRUPT

The on disk structure was partially recognized but the file could not be processed correctly.

Remarks

This function is called within the context of a (post) IRP_MJ_CREATE operation.

This function is called when a file or stream is encountered for the first time; subsequent calls are avoided by caching the result in a stream context. The Provider should inspect the file to see whether it understands the on disk structure and if it does, set *Converted to TRUE, and return the header which was stored with the file when it was converted (see the next section).

If this function does not detect any evidence that it has handled the file it should set *Converted to FALSE and not set the *DtHeader or *DtHeaderLength.

As indicated above, once an On Disk Provider has recognized the file, or stream (by returning *Converted == TRUE) all further activity for this file or stream will be sent to this implementation only.

The *StreamContext which is returned from this function is passed in all these cases, and so can be used to store any per stream data required.

This function has to ensure that it serializes properly. The rest of the FESF system provides no guarantees that this function (or convert below) is not being called for this file/stream in other threads, either on this host or via a networked connection.

If, during its operation, the implementation detects that the file or stream has been partially converted or is in some way inconsistent, it can either fix up the data in situ, or, if this is impossible, returns the distinguished error status DT_CORRUPT.

TeardownStreamObject

```
VOID
TeardownStreamObject(
    _In_ PVOID Object
);
```

This frees up any resources, and deallocates the context returned by DetectState/Convert.

RecoverFile

```
NTSTATUS
RecoverFile(
    _In_     PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _In_     PVOID StreamContext
    );
```

Parameters

FltObjects

This is the information passed in by Filter Manager. The Filter Manager Instance and FileObject are the most useful fields.

VolumeContext

As returned from SetupVolume

StreamContext

As returned fromDetectState or Convert.

Remarks

This would usually be called by On Disk Provider itself in the context of another call to recover state after call to MarkInvalid. The DS Filter calls it only when it needs to know whether a file encrypted

The On Disk Provider should check that the file is in a format it understands, rebuilding any in-memory state from on disk contents, and return STATUS_SUCCESS or a failure status appropriately.

GetHeaders

```
NTSTATUS
GetHeaders(
    _In_ PFLT_INSTANCE Instance,
    _In_opt_ PVOID VolumeContext,
    _In_ PFILE_OBJECT FileObject,
    _In_ PVOID StreamContext,
    _Inout_ PVOID* DtHeader,
    _Inout_ ULONG* DtHeaderSize
   );
```

Input Parameters

Instance
FileObject
The as passed from the Filter Manager and can be used to perform operations on the file.

VolumeContext
As returned from SetupVolume

StreamContext
As returned from DetectState or Convert

Output Parameters

DtHeader
DtHeaderSize
As described below.

Remarks

This function is called within the context of an IRP_MJ_INTERNAL_DEVICE_CONTROL (itself called in post create for the DT) and as a result of the FSCTL_MONADNOCK_DS_READ_DT_HEADER_UNSAFE (potentially called from user mode)

This function is initially invoked by the DT driver component in order to retrieve the DT header contents for the specific file from the DS layer. If the file has no DT Header, this function should return a null pointer and zero length. Otherwise, it allocates a buffer large enough to contain the current DT header and returns a pointer to that buffer as well as the size of that buffer to the caller.

Note that the caller takes responsibility for freeing this buffer by using **ExFreePool** or **ExFreePoolWithTag**. The buffer should be allocated with the tag defined by the manifest constant **TAG_DT_KEY**.

All failure statuses will be reflected up to the calling function, hence this function should return STATUS_SUCCESS even if there is no DT header.

WriteHeader

```
NTSTATUS
WriteHeader(
    _In_ PFLT_INSTANCE Instance,
    _In_opt_ PVOID VolumeContext,
    _In_ PFILE_OBJECT FileObject,
    _In_ PVOID StreamContext,
    _In_ PVOID OldDtHeader,
    _In_ PVOID NewDtHeader,
    _In_ ULONG DtHeaderSize
   );
```

Input Parameters

Instance
FileObject
: The as passed from the Filter Manager and can be used to perform operations on the file.

VolumeContext
: As returned from SetupVolume

StreamContext
: As returned from DetectState or Convert

OldDtHeader
NewDtHeader
DtHeaderSize
: As described below.

Remarks

This optional function is called in response to an FSCTL_MONADNOCK_DS_WRITE_DT_HEADER_UNSAFE Fsctl,

The On Disk Provider should

1. Stabilize its view of the header
2. Check that the length of the header matches the provided length
3. Check that the current header is that provided
4. And everything macthes *atomically* replace the header with the provided on.

It is expected that changing the header in this way will not affect the encryption operations, and no checking to that end is performed. An example of calling this might be when there is a need to change the header (for instance the encrypted key material) on a file which is usually open for the length of the system.

The reference implementation gives an example of its implementation.

Convert and ConvertExisting

```
NTSTATUS
Convert(
    _In_  PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _In_opt_ PVOID DtHeader,
    _In_     ULONG DtHeaderLength,
    _In_     ULONG CipherBlockSize,
    _In_     BOOLEAN ConvertingIO,
    _Out_    PVOID *StreamContext
   );

NTSTATUS
ConvertExisting (
    _In_  PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _In_     PVOID StreamContext,
    _In_opt_ PVOID DtHeader,
    _In_     ULONG DtHeaderLength,
    _In_     BOOLEAN ConvertingIO,
    _In_     ULONG CipherBlockSize
    );
```

Parameters

FltObjects

As passed from Filter Manager, notably giving access to the File Object and Filter Manager Instance.

VolumeContext

As returned from SetupVolume

StreamContext (input or output)

As returned from DetectState

DtHeader

DtHeaderLength

Describe a header which needs to be persisted and returned from calls to DetectState or GetHeaders

ConvertingIO

Will always be FALSE and can be ignored in this release.

CipherBlockSize

As described above.

Returns

Any SUCCESS or FAILURE status, or the special status STATUS_ENCOUNTERED_WRITE_IN_PROGRESS_

Remarks

These functions are called within the context of a (post) IRP_MJ_CREATE operation.

They are called to convert the file into a format which may be recognized by later calls to DetectState. The provider header has to be stored in a way which will allow subsequent calls to DetectState to return it. Convert is called if the file has not been previously detected as having been formatted.

ConvertExisting is called if a file has previously been detected as formatted and the formatting has since been lost (usually because of a destructive create).
If required (because of write sharing), a successful call to these functions may be followed by a call to DetectState (or ReadHeaders) at which point the definitive state of the file will be established.

CipherBlockSize is the size at which blocks are being encrypted. Reads and writes are required to over-read or over-write a block to that boundary, so as to allow decryption. For instance, if the **CipherBlockSize** is 16, then a write of 20 bytes should be rounded out to 32 bytes, and read of 20 bytes should be rounded to 32 bytes. In both cases the length returned should be 20 bytes, but the calling code ensures that the buffers are large enough for these over reads and writes. The DS implementation is responsible for maintaining the **CipherBlockSize** once a file has been converted, either by saving the value or by only supporting one cipher block size. **CipherBlockSize** is guaranteed to be a power of two.

See the sections describing DtCryptoCodeMdl and DtCryptoBlockSize.

The implementation should protect itself against concurrent conversions. If it discovers that a parallel thread (on this or another system) has converted the file it should return the distinguished status STATUS_ENCOUNTERED_WRITE_IN_PROGRESS. This instructs the rest of the system to treat the file as already converted (and reread the header before returning it to the Crypto and policy provider)

GetAttributes

```
VOID
GetAttributes(
    _In_opt_ PVOID VolumeContext,
    _In_opt_ PFILE_OBJECT FileObject,
    _In_ BOOLEAN PagingIo,
    _In_ BOOLEAN ConvertingIo,
    _In_ PVOID StreamContext,
    _Inout_ PDS_ATTRIBUTES Attributes
);
```

Parameters

VolumeContext

As returned from SetupVolume

FileObject

The as passed from the Filter Manager and can be used to perform operations on the file.

PagingIO

If this request is provoked by Paging IO then this will be TRUE. All reads to the file should be marked IO_PAGING.

ConvertingIO

Will always be FALSE and can be ignored in this release.

StreamContext

As returned from DetectState

Attributes

Attributes about the file in question

Remarks

This function is used to get attributes about the file in question. Often, but not always, it is in response to IRP_QUERY_INFORMATION or a IRP_QUERY_DIRECTORY_INFORMATION.

The PDS_ATTRIBUTES structure contains three lengths (AllocationSize, EndOfFile, ValidDataLength) which should all be returned.

The ValidDataLength is the “high water mark” that data has been written to disk, it can often be derived from the FSD’s ValidDataLength. Unfortunately there is no supported way of getting this value but usually inspecting the FSD’s Advanced FsRtl Header will provide an appropriate value.

Finally, the PDS_ATTRIBUTES also contains a “UserDataOffset”. If the user data is going to be written at a different offset from the one that the user sees, then this should be the offset that will be applied. DS uses this to ensure that the FSD acquires the correct parameters when called at many of the places where it needs an offset (for instance acquire for Modified or Mapped page write, set sparse and so forth).

IsPotentiallyEncrypted

```
BOOLEAN
IsPotentiallyEncrypted (
      PLARGE_INTEGER Length
      );
```

There are several cases during create and directory size correction when significant processing can be avoided if we know a priori that the file under question cannot be encrypted – either because its length is unreasonable, or it is too small.

This required entry point allows an on disk implementation to instruct the DS infrastructure to avoid these expensive operations. If a specific On Disk Structure implementation cannot make the call it should always return TRUE.

PreReadWrite, PreSetInfo

```
FLT_PREOP_CALLBACK_STATUS
PreReadWrite(
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_    PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _In_ PVOID StreamContext
    );

FLT_PREOP_CALLBACK_STATUS
PreSetInfo(
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_    PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _In_ PVOID StreamContext
    );
```

Parameters

Data
FltObjects
As provided by Filter Manager.

VolumeContext
As returned from SetupVolume

StreamContext
As returned from DetectState or Convert

Remarks

These functions are called in the relevant Filter Manager callback as soon as the designated component has been located. Just as in the Filter Manager case, the returned value affects continued processing of the operation and whether the `DS_POST_CALL` will be made.

Notes on Cipher Blocksize

You need to be aware that the sizes involved for reads and writes and setting of length are those required by the application. However the Encryption Provider implementation will often require that

more data be stored in order that the user data be correctly decrypted. The rounding is that specified as `CipherBlockSize` in the call to `Convert`.

If a read or write size is not aligned on a cipher block boundary, the provided buffer *will* be large enough for the data up to the cipher blocked rounded boundary.

So if a read is for 100 bytes, but the Cipher Block size was specified as being 16, then the actual amount of data read should be 112 bytes.

PostReadWrite, PostSetInfo

```
FLT_POSTOP_CALLBACK_STATUS
PostReadWrite(
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_    PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _In_ PVOID StreamContext
    );

FLT_POSTOP_CALLBACK_STATUS
PostSetInfo(
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_    PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID VolumeContext,
    _In_ PVOID StreamContext
    );
```

Parameters

Data

FltObjects

As provided by Filter Manager.

VolumeContext

As returned from SetupVolume

StreamContext

As returned from DetectState or Convert

Remarks

This function is called in the relevant Filter Manager callback.

IsValid

```
BOOLEAN
IsValid(
    _In_ PVOID StreamContext
)
```

Parameters

StreamContext

Remarks

Do we currently believe this file to be valid?

MarkInvalid

```
VOID
MarkInvalid(
    _In_ _Notnull_ PVOID VolumeContext,
    _In_ _Notnull_ PFILE_OBJECT FileObject,
    _In_ _Notnull_ PVOID StreamContext
    );
```

Parameters

VolumeContext

FileObject

StreamContext

Remarks

Something has happened to mark the file invalid.

LastWriteableCleanup

```
VOID
LastWriteableCleanup (
    _In_    PCFLT_RELATED_OBJECTS FltObjects,
    _In_    PVOID VolumeContext,
    _In_    PVOID StreamContext,
    _In_    BOOLEAN IsConvertingPagingIo
    );
```

Parameters

FltObjects

As provided by Filter Manager.

VolumeContext

As returned from SetupVolume

StreamContext

As returned from DetectState or Convert

IsConvertingPagingIo

Should be ignored and will always be set to FALSE

Remarks

This is called in pre-cleanup when the number of (non raw) open handles opened for write on the file has gone to zero. Typically an On Disk Provider might use this to make sure that the file is written to EOF (thus avoiding re-entrant writes).

NOTE for certain file system which check VDL against EOF in every cleanup path (an example at time of writing is ExFat) this may be called for all closes of (non raw) handles opened for Write.

LastWriteableClose

```
VOID
LastWriteableClose (
    _In_    PCFLT_RELATED_OBJECTS FltObjects,
    _In_    PVOID VolumeContext,
    _In_    PVOID StreamContext,
    _In_    BOOLEAN IsConvertingPagingIo
    );
```

Parameters

FltObjects

As provided by Filter Manager.

VolumeContext

As returned from SetupVolume

StreamContext

As returned from DetectState or Convert

IsConvertingPagingIo

Should be ignored and will always be set to FALSE

Remarks

This is called in pre-close when the number of (non raw) file objects opened for write on the file has gone to zero. Typically an On Disk Provider might use this to quiesce the on disk structure, for instance to write checkpoint clean records.

On Disk Support Functional Reference

Directory Correction Support

Header: `FESFDirCorrection.h`

Lib: `Misc.lib`

FesfGetNextDirectoryEntry

```
PVOID
FesfGetNextDirectoryEntry(
      _In_ PVOID Buffer,
      _In_ FILE_INFORMATION_CLASS InfoClass,
      _Inout_ PULONG SizeLeft,
      _Inout_bytecount_(sizeof(FESF_DIRECTORY_ENTRY)) PFESF_DIRECTORY_ENTRY
Addresses);
```

Parameters

Buffer

The current buffer pointer. Initially it should be populated with the Buffer pointer provided to `DtSupportPerFileCorrect`. On subsequent calls it should be the value returned from the previous call.

InfoClass

As pointer provided to DtSupportPerFileCorrect

SizeLeft

A pointer to a ULONG initially populated with the size of the buffer provided to `DtSupportPerFileCorrect`

Addresses

The address of a fixed size buffer which receives the information about the next entry.

Returns

This function returns NULL if the enumeration failed or if the end of the buffer has been reached. If this function fails because of invalid input, then the **FileName.Buffer** field will be NULL.

Remarks

This function is a helper for `DtSupportPerFileCorrect` and abstracts away the differences between different directory formats by conditionally populating a fixed structure with pointers to the various possible fields in a directory entry. After a successful call, the **FileName** and **Attributes** fields of the Addresses parameter will be filled in. All other fields will be optionally filled in.

FesfIsMarkedForCorrection

```
BOOLEAN
FesfIsMarkedForCorrection(
      _In_bytecount_(sizeof(FESF_DIRECTORY_ENTRY)) PFESF_DIRECTORY_ENTRY
Addresses
);
```

Parameters

Addresses

Contains a buffer returned by `FesfGetNextDirectoryEntry.`

Returns

TRUE if the data provider has indicated that the entry is for a file which may require correction.

FesfClearForCorrection

```
VOID
FesfClearForCorrection(
    _Inout_bytecount_(sizeof(FESF_DIRECTORY_ENTRY)) PFESF_DIRECTORY_ENTRY
Addresses);
```

Parameters

Addresses

Contains a buffer returned by `FesfGetNextDirectoryEntry.`

Remarks

This function instructs the data provider to not apply correction, thus returning the “raw size” of the directory entry to the enumerating application.

Process and Thread Support

Header: `PsSup.h`

Lib: `OsrSupport.lib`

Contains several support functions. Only one is liable to be of use.

OsrGetProcessImageName

```
NTSTATUS
OsrGetProcessImageName(
    _In_ PEPROCESS Process,
    _Inout_ PUNICODE_STRING ProcessImageName
);
```

Parameters

Process

Refers to the process whose image is to be returned.

ProcessImageName

Describes where the return value should be placed. The space allocated (described by `ProcessImageName->Buffer` and `ProcessImageName->Length`) must be large enough for the return string.

Returns

STATUS_SUCCESS if the `ProcessImageName` has been successfully set up.
STATUS_BUFFER_OVERFLOW if the buffer described by `ProcessImageName` is too small. In this case `ProcessImageName->Length` is set to the required length.
Other failure statuses can be returned.

Remarks

This function returns the image name associated with the provided process.