

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Securing a computer system entails employing measures that protect the computer's data from viewing or manipulation by unauthorized users. Security measures at the network interface prevent intruders from gaining entry to the computer, and file-system security prevents the computer's authorized users from accessing data they're not supposed to access. However, a computer that is isolated from the Internet behind a firewall and that has stringent file-system security policies in place remains unsecured if no strategy exists to guard the computer's physical security. If unauthorized users have physical access to a computer, they can remove the computer's hard disks and perform offline analysis of the disks' data. When users can view a hard disk's contents on a different computer, file-system security (e.g., the kind NTFS ACLs provide on Windows NT or Windows 2000—Win2K—systems) is of no value. This problem is especially acute for laptop computers because two NTFS file-system drivers that ignore NTFS security—NTFSDOS and an NTFS driver for Linux—let even casual thieves easily view NTFS files.

One way to address the physical security problem is to keep computers in locked rooms, but this solution is obviously not practical for laptop computers, whose main purpose is portability. Thus, to prevent access to file data in situations in which bypassing file-system security is a possibility, data encryption is necessary. Before Win2K, NT users have had to turn to third-party vendors for encryption solutions, but in Win2K a built-in encryption facility for NTFS files exists in the form of Encrypting File System (EFS). By building encryption into the OS, Microsoft can make the encryption and decryption process transparent to both applications and users.

Unfortunately, Microsoft has produced little documentation describing how EFS works. Because many people will undoubtedly rely on EFS to secure their sensitive data, having a solid understanding of what goes on under the hood is important. In this two-part series about EFS, I'll take you beneath the surface and let you know exactly how EFS works with NTFS and Win2K cryptography facilities to help you keep your data safe from prying eyes. This month, I provide an overview of EFS and begin walking you through the process by which EFS encrypts files. Next month, I'll finish the encryption walk-through, describe the decryption process, and introduce the data recovery mechanism EFS has built into the decryption process.

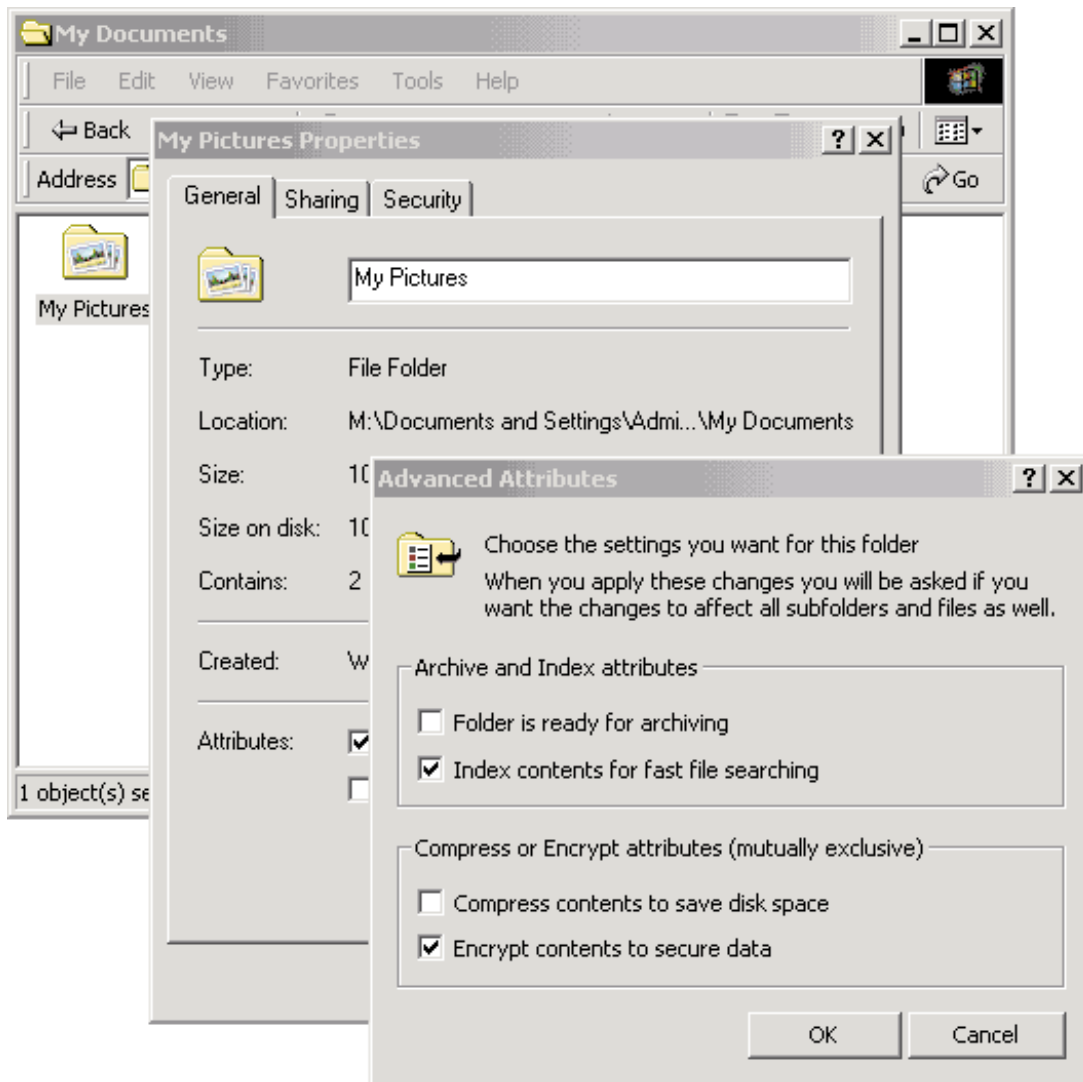
## EFS Overview

EFS security relies on Win2K cryptography support, which Microsoft introduced in NT 4.0. The first time you encrypt a file, EFS assigns your account one public key and private key pair for use in file encryption. You can deliberately encrypt files via an NT Explorer dialog box, as Screen 1, page 52, shows, or a command-line utility. Win2K automatically encrypts files that reside in directories the OS designates as encrypted directories. When you encrypt a file, EFS generates a random number for the file that EFS calls the file's file encryption key (FEK). EFS uses the FEK to encrypt the file's contents with a stronger variant of the Data Encryption Standard (DES) algorithm—DESX. EFS stores the file's FEK with the file but encrypts the file with your EFS public key using the RSA public key-based encryption algorithm. After EFS completes these steps, the file is secure: Other users can't decrypt your data without the file's decrypted FEK, and they can't decrypt the FEK without your private key.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)



Why does EFS use a public key/private key algorithm to encrypt FEKs, and DESX to encrypt file data? Because DESX uses the same key to encrypt and decrypt data, it is a symmetric encryption algorithm. Symmetric encryption algorithms are typically very fast, which makes them suitable for encrypting large amounts of data, such as file data. However, symmetric encryption algorithms have a weakness: You can bypass their security if you obtain the key. If multiple users want to share one encrypted file protected only by DESX, each user would require access to the file's FEK. Leaving the FEK unencrypted would obviously be a security problem, but encrypting the FEK once would require all the users to share the same FEK decryption key—another potential security problem.

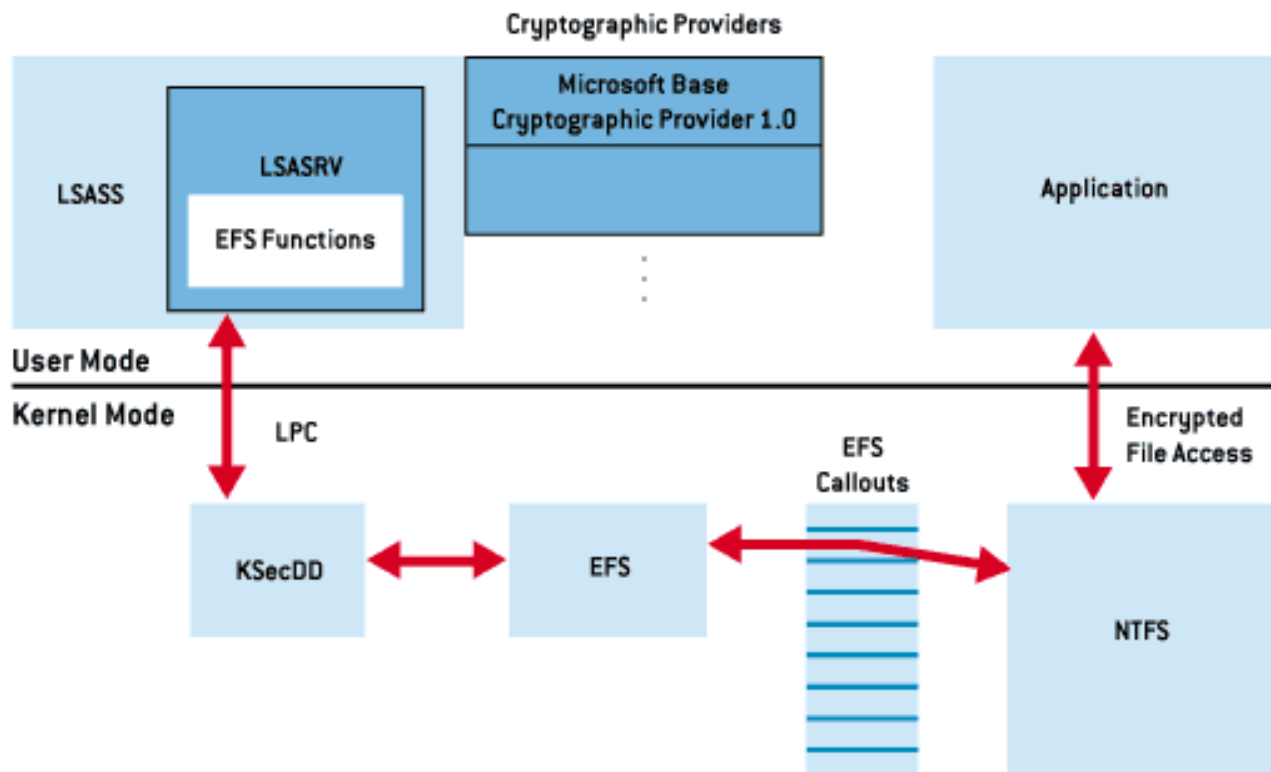
Keeping the FEK secure is the difficult problem EFS addresses with the public-key-based half of its encryption architecture. Encrypting a file's FEK for individual users who access the file lets multiple users share an encrypted file. EFS can encrypt a file's FEK with each user's public key and can store each user's encrypted FEK with the file. Anyone can access a user's public key, but no one can use a public key to decrypt the data that the public key encrypted. The only way users can decrypt a file is with their private key, which the OS must access and typically stores in a secure location. A user's private key decrypts the user's encrypted copy of a file's FEK. Win2K's first release will store private keys on a computer's hard disk (an arrangement that isn't terribly secure), but subsequent releases of

# Inside Encrypting File System

Mark Russinovich  
(Reprinted from WindowsItPro Magazine)

the OS will let you store your private key on a smart card, for example. Public-key-based algorithms are usually slow, but EFS uses these algorithms only to encrypt FEKs. Splitting key management between a publicly available key and a private key makes key management a little easier than symmetric encryption algorithms do, and solves the dilemma of keeping the FEK secure.

Several components work together to make EFS work, as the diagram of EFS architecture in Figure 1 shows. EFS is a device driver that runs in Win2K's kernel mode, in which EFS is tightly connected with the NTFS file-system driver. Whenever NTFS encounters an encrypted file, NTFS executes functions in the EFS driver that the EFS driver registered with NTFS when EFS initialized. The EFS functions encrypt and decrypt file data as applications access encrypted files. Although EFS stores a FEK with a file's data, users' public keys encrypt the FEK. To encrypt or decrypt file data, EFS must decrypt the file's FEK with the aid of cryptography services that reside in user mode.



The Local Security Authority Subsystem (lsass.exe) manages logon sessions but also handles EFS key management chores. For example, when the EFS driver needs to decrypt a FEK to decrypt file data a user wants to access, the EFS driver sends a request to LSASS. EFS sends the request via local procedure call (LPC), a form of interprocess communication in NT and Win2K that the remote procedure call (RPC) API uses for communication between two endpoints on the same computer. (To read more about LPC and RPC, see NT Internals: "Inside NT Networking," March 1999.) The ksecdd.sys driver exports functions for other drivers that need to send LPC messages to LSASS. The Local Security Authority Server (lsasrv.dll) component of LSASS that listens for RPC requests passes requests to decrypt a FEK to the appropriate EFS-related decryption function, which also resides in LSASRV. LSASRV uses functions in Microsoft's CryptoAPI to decrypt the FEK, which the EFS driver sent to LSASS in encrypted form. The CryptoAPI comprises Cryptographic Provider DLLs that make various cryptography services (such as encryption/decryption and hashing) available to applications.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

For example, the Cryptographic Provider DLLs manage user private and public key storage and retrieval so that LSASRV doesn't need to concern itself with the details of the encryption algorithms or how keys are protected. After LSASRV decrypts a FEK, LSASRV returns the FEK to the EFS driver via an LPC reply message. After EFS receives the decrypted FEK, EFS can use DESX to decrypt the file data for NTFS. Let's look at the details of how EFS integrates with NTFS, and how LSASRV uses the CryptoAPI to manage keys.

## Registering Callbacks

NTFS doesn't require the EFS driver's presence to execute, but encrypted files won't be accessible if the EFS driver isn't present. NTFS has a plug-in interface for the EFS driver, so when the EFS driver initializes, it can attach itself to NTFS. The NTFS driver exports several functions for the EFS driver to use, including the NtOfsRegisterCallbacks function. EFS calls NtOfsRegisterCallbacks to notify NTFS both of EFS's presence and of the EFS-related APIs EFS is making available. All of the other functions NTFS exports for the EFS driver begin with the prefix NtOfs, which presumably stands for NT Object File System. One of the original goals of the NT 5.0 project (code-named Cairo) was to develop an object-oriented file system. Although NTFS in Win2K probably hasn't reached the level of object orientation that the Cairo planners had in mind, Microsoft has extended NTFS in several significant ways from its NT 4.0 implementation. One of those ways is NTFS's support for encrypted files via the NtOfs interfaces.

EFS registers the seven EFS APIs that Table 1 lists with NTFS via the NtOfsRegisterCallbacks interface. So that EFS can provide transparent support for encrypted files, at appropriate places throughout NTFS code execution the NTFS driver looks into the table of registered callbacks and invokes the callback that EFS must execute at that point. NTFS supports only one set of registered callbacks; in other words, other device drivers can't use NtOfsRegisterCallbacks because NTFS assumes that only the EFS driver will plug in to the callback interface. You'll understand better how EFS can perform encryption and decryption based on its callback invocation from NTFS as I walk you through the encryption and decryption process.

TABLE 1: EFS Callbacks	
Callback Function	When NTFS Invokes the Callback
EfsOpenFile	When an application opens an existing file
EfsFilePostCreate	After an NTFS file has created or opened a file for an application
EfsFileControl	When a user modifies a file's encryption settings
EfsFsControl	When a user modifies a file's encryption settings
EfsRead	When NTFS retrieves data for an application
EfsWrite	After an application writes file data
EfsFreeContext	When context data buffer is no longer needed

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

## Encrypting a File for the First Time

The NTFS driver calls only the EFS functions that register when NTFS encounters an encrypted file. A file's attributes record that the file is encrypted in the same way that a file records that it's compressed. NTFS and EFS have specific interfaces for converting a file from nonencrypted to encrypted form, but user-mode components primarily drive the process. Win2K lets you encrypt a file in two ways: by using the Cipher command-line utility or by checking the Encrypted check box in the Advanced properties dialog box for a file in NT Explorer. Both NT Explorer and Cipher rely on the new EncryptFile Win32 API that the Win32 DLL—advapi32.dll (Advanced Win32 APIs DLL)—exports. ADVAPI32 loads another DLL, feclient.dll (File Encryption Client DLL), to obtain APIs that ADVAPI32 can use to invoke EFS interfaces in LSASRV via LPC.

When LSASRV receives an LPC message from FECLIENT to encrypt a file, LSASRV invokes the internal function EfsRpcEncryptFileSrv. EfsRpcEncryptFileSrv's first step is to use Win2K's impersonation facility to impersonate the user that ran the application (either Cipher or NT Explorer) that is encrypting the file. This procedure lets Win2K treat the file operations that LSASRV performs as if the user who wants to encrypt the file is performing them. LSASRV usually runs in the System account, which has different (and more privileged) security capabilities than a typical user account has. If it doesn't impersonate the user, LSASRV might not have permission to access the file in question.

The next step EfsRpcEncryptFileSrv performs is to create a log file into which LSASRV will record the progress of the encryption process. EfsRpcEncryptFileSrv creates the log file on the same drive as the file that EFS will encrypt, and places the log file under the root directory's subdirectory System Volume Information. The log file usually has the name efs0.log, but if other files are undergoing encryption, EfsRpcEncryptFileSrv replaces the number 0 with increasing numbers until LSASRV can create a unique log file for the current encryption.

Win2K's cryptography APIs rely on information that a user's Registry profile stores, so EfsRpcEncryptFileSrv next uses the LoadUserProfile API of userenv.dll (User Environment DLL) to load the profile into the Registry of the user EfsRpcEncryptFileSrv is impersonating. Usually the user profile is already loaded, because winlogon.exe loads a user's profile when a user interactively logs on. However, if you use the Microsoft Windows NT Server 4.0 Resource Kit Su utility or the Win2K RunAs command to log on to a different account, when you try to access encrypted files from that account, the account's profile might not load.

EfsRpcEncryptFileSrv's next step is to call another LSASRV function, EncryptFileSrv, to carry out the rest of the encryption process for the file. EncryptFileSrv begins by querying NTFS about which data streams exist within the file, noting the result for later use. NTFS supports alternate named data streams in addition to the default unnamed stream in which NT typically stores data. Win2K makes heavy use of alternate streams to support compound document storage (i.e., Native Structured Storage), as does Services for Macintosh, which uses alternate streams to implement Macintosh resource forks. EFS must encrypt all of a file's data streams—not just the stream that most applications see. EncryptFileSrv calls the internal function GenerateFEK to generate a FEK for the file. GenerateFEK uses the CryptoAPI function CryptAcquireContext to initiate a cryptographic session.

CryptAcquireContext takes several parameters, including the name of the session's cryptographic provider and the cryptography service that the caller (in this case, GenerateFEK) is interested in. GenerateFEK specifies Microsoft Base Cryptographic Provider 1.0 as the session's cryptographic provider. GenerateFEK also signals that it wants to use the provider's RSA encryption facilities. Base Cryptographic Provider is a built-in provider that is present on all Win2K systems. However, the architecture of the CryptoAPI lets software vendors implement proprietary providers and dynamically

# Inside Encrypting File System

Mark Russinovich

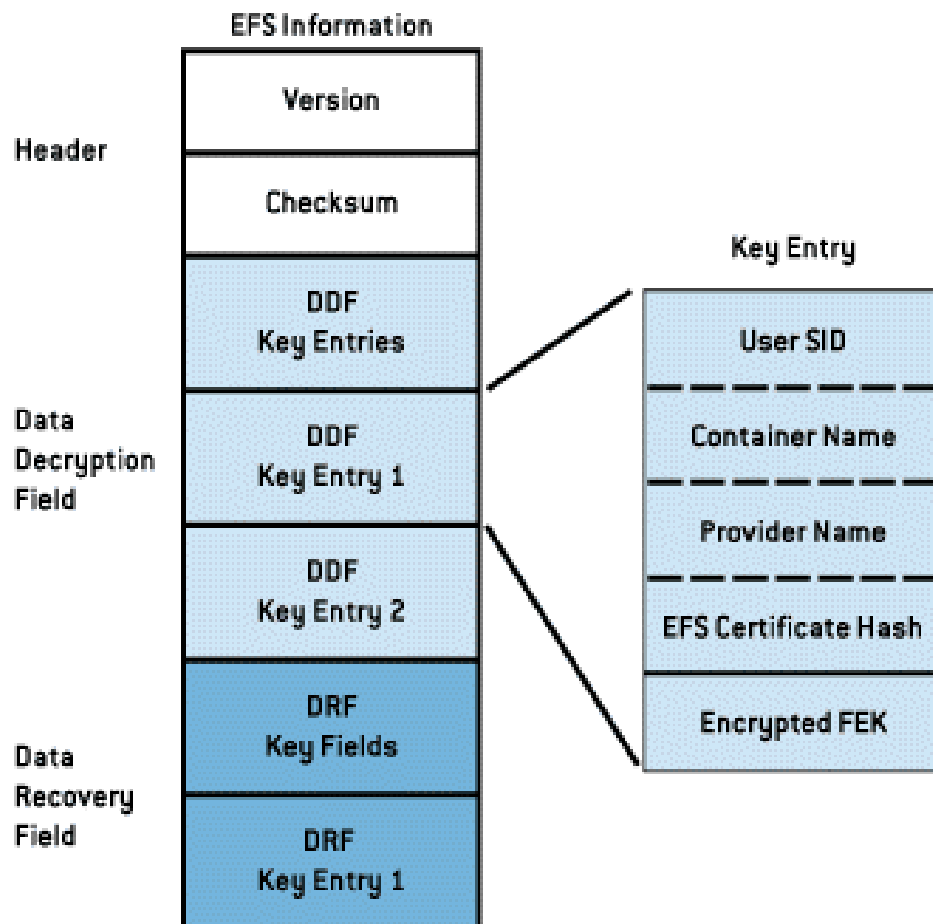
(Reprinted from WindowsItPro Magazine)

add them to Win2K. Thus, GenerateFEK must specify a session's cryptographic provider. RSA is a public-key-based encryption algorithm that has become a de facto worldwide standard. The file rsabase.dll implements RSA for Base Cryptographic Provider 1.0.

After the CryptAcquireContext function returns with a handle to the provider, GenerateFEK calls CryptGenRandom to have the provider generate 16 bytes (128 bits) of random data to serve as the file's FEK. GenerateFEK then calls CryptCloseSession to close the cryptographic provider session, and returns control to EncryptFileSrv.

## Constructing Key Rings

At this point, EncryptFileSrv has a FEK and can construct EFS information to store with the file, including an encrypted version of the FEK. Figure 2 illustrates the EFS information's layout. EncryptFileSrv calls another function, ConstructEFS, to construct the EFS information. Before it can do so, ConstructEFS must use the CryptoAPI to get a handle to the user's public key and private key pair. To get this handle, ConstructEFS calls another function, GetCurrentKey, which reads the Registry value HKEY\_CURRENT\_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash. Every public key/private key pair has a digital certificate that the issuing certificate authority signs, and that users use to obtain their public keys. The digital signature, or hash, uniquely identifies the public key/private key pair. By reading the CertificateHash value, ConstructEFS obtains the current user's public key signature and uses it to access the public key and encrypt FEKs.



# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

When the CertificateHash value doesn't exist, which is the case the first time a user encrypts a file, EFS must determine whether the user has an EFS public key/private key pair, or whether it must create the key pair. First, EFS opens the My system certificate storage area, in which the OS stores EFS key pairs. (Several certificate storage areas can exist on a system, each containing various certificates.) EFS uses the CryptoAPI function CertFindCertificateInStore to look for an EFS key-pair certificate in a provider's My storage area. If CertFindCertificateInStore doesn't find an EFS key-pair certificate, ConstructEFS calls GenerateUserKey to create one. GenerateUserKey calls CryptUIWizCertRequest to create the key pair and return a signed certificate for the pair. The key-pair generation occurs on a domain controller (for a system that is part of a domain) or the local computer (for a computer that is not part of a domain). When EFS locates or creates the user's key-pair certificate, EFS obtains the certificate's hash and stores it in the Registry key HKEY\_CURRENT\_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash. In Win2K beta 3, the provider that creates public key/private key pairs is Base Cryptographic Provider 1.0.

When it has the user's key-pair hash, EFS uses the CertGetCertificateContextProperty CryptoAPI to obtain information about the provider that CACreateLocalAutoEnrollmentObject used to generate the key pair. This information includes the provider's name and the name of the container the provider uses to store key pairs for the user whom LSASRV is impersonating. The container name is meaningful only to the provider, but in the case of Base Cryptographic Provider, the container name is a file path relative to the user's profile directory. An example container name is M:\Documents and Settings\Administrator\Application Data\Microsoft\SystemCertificates\My\Certificates\CD099602FD898D7EFCDC4283C6742D30A15A0062. Win2K hides the Application Data directory by default, and container filenames vary. EFS uses CryptAcquireContext to open a cryptographic session with the provider. In the call to CryptAcquireContext, EFS specifies the provider name, the container name, and that it wants to use RSA encryption services. To obtain the current user's public key, EFS uses the function CryptExportKey, which causes the cryptographic provider to extract the key from the container.

ConstructEFS can now construct the information that EFS stores with the file. The LSASRV function ConstructKeyRing makes use of the user's public key that CryptExportKey obtained to store the EFS information with a file. Microsoft calls the function ConstructKeyRing because, as I mentioned earlier, EFS lets multiple users share encrypted files. EFS stores only one block of information in an encrypted file, and that block contains an entry for each user sharing the file. These entries are called key entries, and EFS stores them in the Data Decryption Field (DDF) portion of the file's EFS data. A collection of multiple key entries is a key ring.

Figure 2 shows a file's EFS information format and key entry format. EFS stores enough information in the first part of a key entry to precisely describe a user's public key. This data includes the cryptographic provider name, the container name in which the key is stored, the user's security ID (SID), and the public key/private key pair certificate hash. The second part of the key entry contains an encrypted version of the FEK. ConstructKeyRing uses the CryptEncrypt CryptoAPI to encrypt the FEK with the RSA algorithm and the user's public key. (When I describe the decryption process next month, I'll explain why EFS stores this information in key entries.)

Next, EFS creates another key ring that contains recovery key entries. EFS stores information about recovery key entries in a file's Data Recovery Field (DRF). The format of DRF entries is identical to the format of DDF entries. The DRF's purpose is to let designated accounts, or Recovery Agents, decrypt a user's file when administrative authority must have access to the user's data. For example,

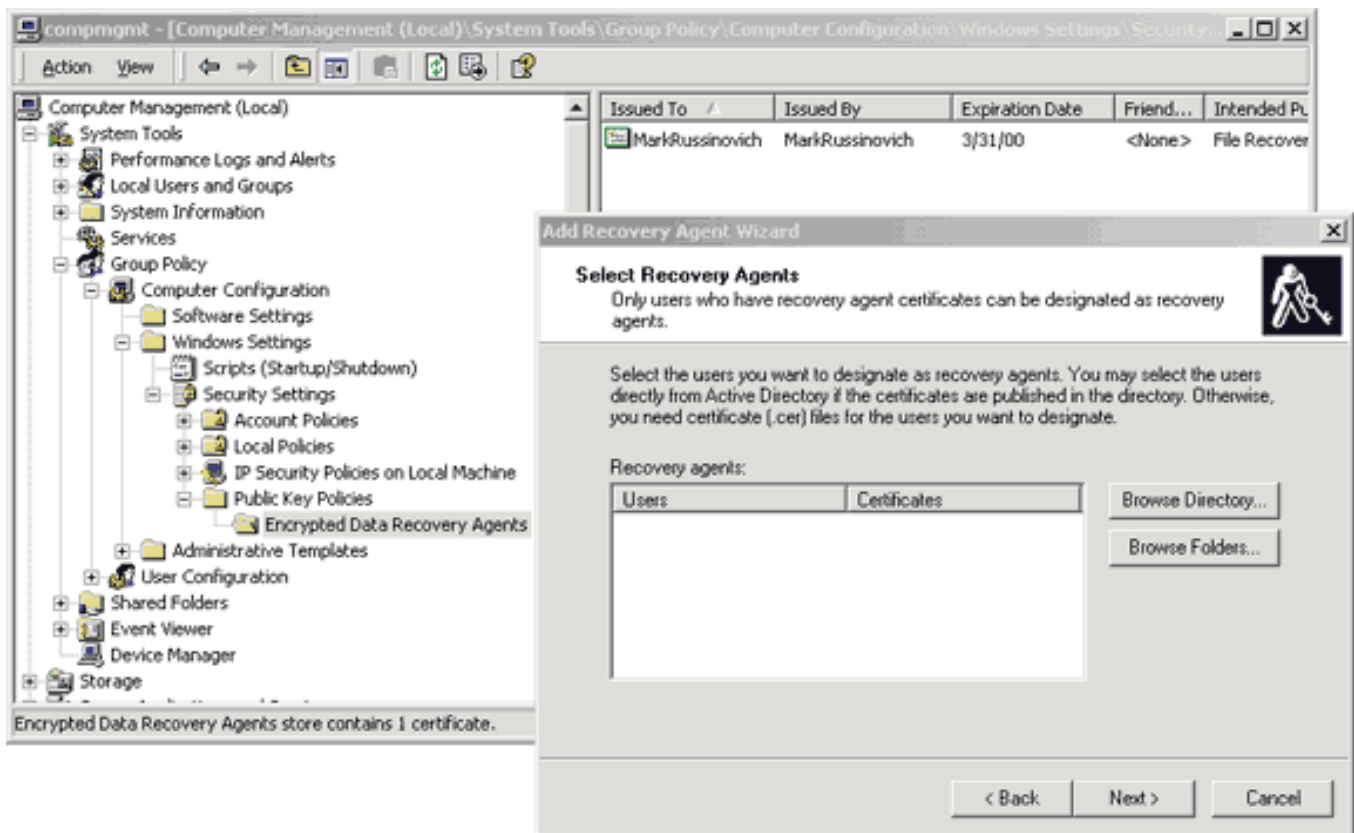
# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

suppose a company employee used a CryptoAPI to store his private key on a smart card, then lost the card. Without Recovery Agents, no one could recover his encrypted data.

You can define Recovery Agents with the Encrypted Recovery Agents security policy (which is a subset of public key policy) of the local computer or domain, as Screen 2 shows. When you use the Recovery Agent Wizard in the domain or computer management Microsoft Management Console (MMC) snap-in, you can add Recovery Agents and specify which public key/private key pairs (designated by their certificates) the agents use for EFS recovery. LSASRV interprets the recovery policy when it initializes and when it receives notification that the recovery policy has changed. EFS uses the cryptographic provider you register for EFS recovery to create a DRF key entry for each Recovery Agent. Currently, the default Recovery Agent provider is the RSA encryption facility of Base Cryptographic Provider 1.0—the same provider GenerateFEK uses for user keys.

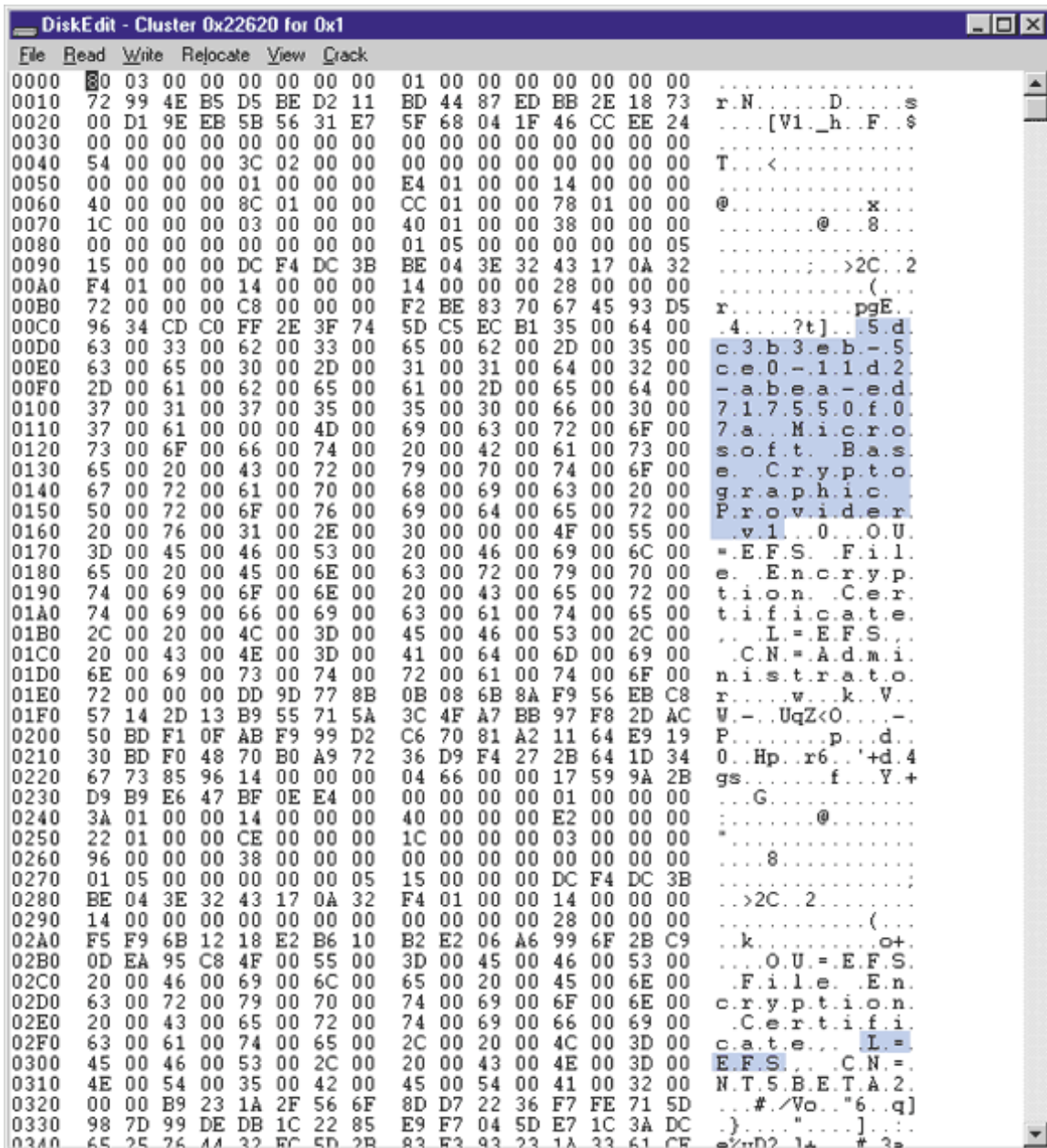


In the final step in creating EFS information for a file, ConstructEFS uses the MD5 hash facility of Base Cryptographic Provider 1.0 to calculate a checksum for the DRF and DDF. EFS stores the checksum's result in the EFS information header. EFS references this checksum during decryption to ensure that the contents of a file's EFS information have not become corrupted or been tampered with. Screen 3 shows an encrypted file's \$EFS attribute's on-disk contents. Some of the fields I've described are highlighted, including the name of the cryptographic store that stores the key pair the encryption uses (the string beginning 5.d.c.3.b.3.e.b.-.5.), the name of the cryptographic provider (Microsoft Base Cryptographic Provider v1.), and information used for display when applications query the encryption format (L.=.E.F.S.). DiskEdit, the utility viewing the data in Screen 3, is an unsupported tool that Microsoft includes on the Service Pack 4 (SP4) CD-ROM.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)



That brings us to the end of this month's column. Next month, I'll conclude my description of the encryption process, and I'll describe the file decryption process and some of the other EFS APIs that LSASRV supports.

Last month, I began this two-part series by introducing the basics of Encrypting File System (EFS). I concluded by discussing how EFS generates keys and stores the keys in an EFS attribute with the file the keys will encrypt. This month, I conclude my walk through the encryption process. I also discuss the decryption process and other functionality that the EFS driver provides, including encrypted file backup and restore and the ability to view information about encrypted files.

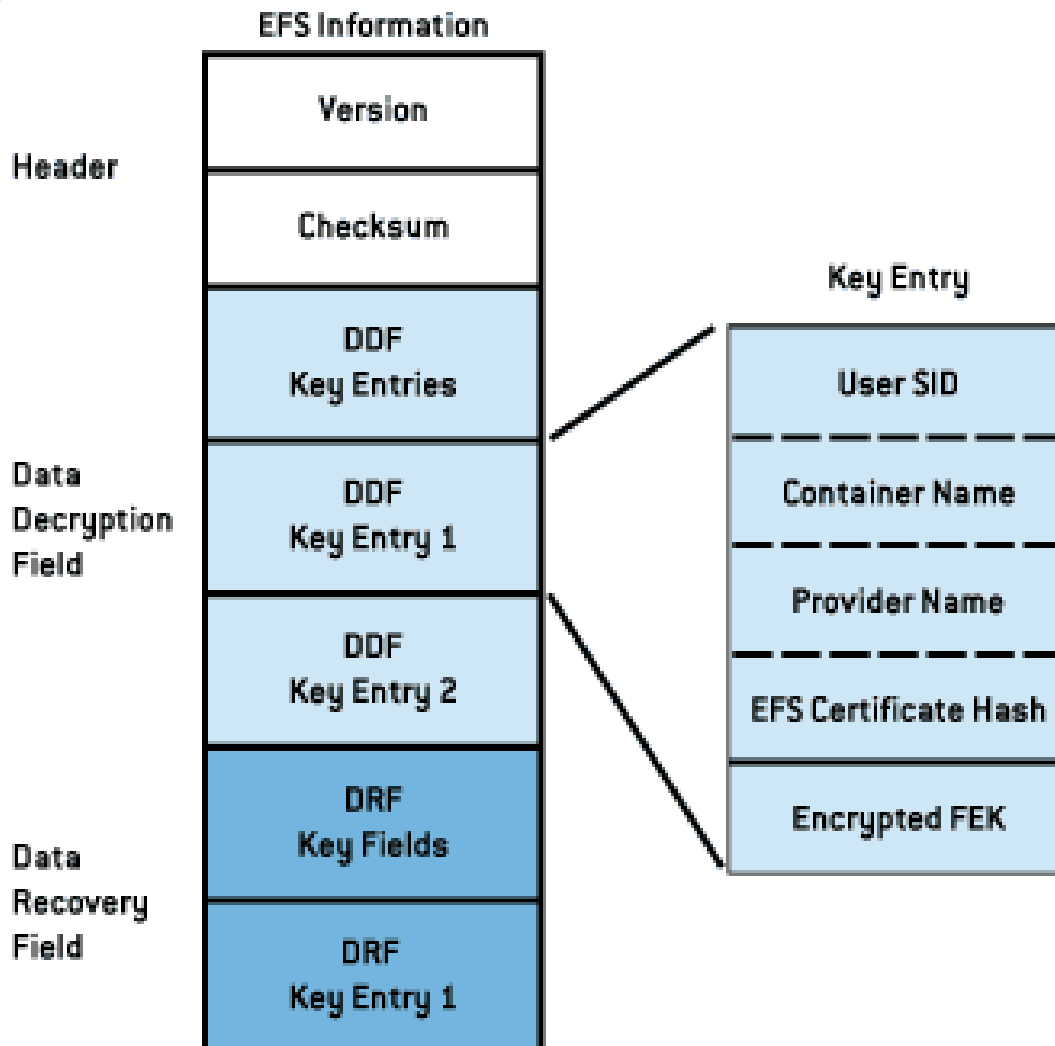
I concluded Part 1 at the point at which EFSEncryptFileSrv finishes creating Data Decryption Field (DDF) and Data Recovery Field (DRF) key fields for a file that is undergoing encryption. Figure 1,

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

page 54, depicts the encryption process' flow. After EfsEncryptFileSrv constructs the necessary information for a file a user wants to encrypt, EFS can begin encrypting the file. The Local Security Authority Server's (LSASRV's) EncryptFileSrv function guides this phase of the encryption process. First, EncryptFileSrv creates a backup file, efs0.tmp, for the file undergoing encryption. (EncryptFileSrv uses higher numbers in the backup filename if other backup files exist.) EncryptFileSrv creates the backup file in the directory that contains the file undergoing encryption. Then, EFS applies a restrictive security descriptor to the backup file so that only the OS (i.e., the System account) can access the file's contents. EFS initializes the log file that EfsRpcEncryptFileSrv created in the first phase of the encryption process. Finally, EFS records in the log file that EncryptFileSrv created the backup file. EFS encrypts the original file only after the file is completely backed up.



EncryptFileSrv next prepares to send the EFS device driver a command to add to the original file the EFS information that EncryptFileSrv just created. EncryptFileSrv encrypts the command with DESX (a stronger variant of Data Encryption Standard—DES) and a session key before sending the command. When the system initializes, LSASRV uses the CryptoAPI function CryptGenRandom to produce a 128-bit number to serve as the session key. The EFS driver asks LSASRV for the session key via local procedure call (LPC); when the driver has the key, the driver can decrypt the control commands that EncryptFileSrv encrypted with the session key. EFS encrypts control commands so that malicious

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

users can't develop programs to send commands specifying that files adopt incorrect EFS information, or that destroy files' valid EFS information. (A file with invalid EFS information is essentially destroyed because EFS can't then interpret the file's data correctly.)

NTFS receives the command to add the EFS information to the original file, but because NTFS doesn't understand EFS commands, NTFS calls the EFS driver function `EfsFileControl`. `EfsFileControl` uses its copy of the session key to decrypt the control command, then calls the `EfsSetEncrypt` function. `EfsSetEncrypt` takes the EFS information that LSASRV sent and uses exported NTFS functions to apply the information to the file. The exported NTFS functions let EFS add meta data information to NTFS files. The EFS driver uses `NtOfsCreateAttributeEx`, `NtOfsSetLength`, `NtOfsPutData`, and `NtOfsCloseAttribute` to create the \$EFS NTFS meta data attribute (which is new to Windows 2000—Win2K) and copy the EFS information to the attribute. Simultaneously, the EFS driver returns a block of context data to NTFS and associates the data with the file's NTFS in-memory management data. NTFS will pass this data, or context information, to EFS whenever NTFS invokes an EFS driver callback function for the file. EFS makes use of the context information to store an unencrypted version of the file's file encryption key (FEK).

Execution returns to `EncryptFileSrv`, which copies the contents of the file undergoing encryption to the backup file. When the backup copy is complete, including backups of all alternate data streams, `EncryptFileSrv` records in the log file that the backup file is up-to-date. `EncryptFileSrv` then sends another DESX-encrypted command to NTFS to tell NTFS to encrypt the contents of the original file.

When NTFS receives the EFS command to encrypt the file, NTFS deletes the contents of the original file and copies the backup data to the file. After NTFS copies each section of the file, NTFS flushes the section's data from the file system cache, which prompts the Cache Manager to tell NTFS to write the file's data to disk. Because the file is marked as encrypted, at this point in the file-writing process, NTFS calls EFS to encrypt the data before NTFS writes the data to disk. The EFS function `EfsWrite` uses the unencrypted FEK in the file's context information to perform DESX encryption of the file, one sector (512 bytes) at a time.

On Win2K versions approved for export outside the United States, the EFS driver implements a 56-bit key DESX encryption. I stated in Part 1 that a FEK is 128 bits long; however, only the first 56 bits constitute the DESX key for the EFS export version. For the US-only version of Win2K, the key is 128 bits long, and the entire FEK constitutes the key.

After EFS encrypts the file, `EncryptFileSrv` records in the log file that the encryption was successful and deletes the file's backup copy. Finally, `EncryptFileSrv` deletes the log file and returns control to the application that requested the file's encryption.

Table 1 summarizes the steps EFS performs to encrypt a file. If the system crashes during the encryption process, either the original file remains intact or the backup file contains a consistent copy. When LSASRV initializes after a system crash, it looks for log files under the System Volume Information subdirectory on each NTFS drive on the system. If LSASRV finds one or more log files, it examines their contents and determines how recovery should take place. LSASRV deletes the log file and the corresponding backup file if the original file wasn't modified at the time of the crash; otherwise, LSASRV copies the backup file over the original, partially encrypted file, then deletes the log and backup. After EFS processes log files, the file system will be in a consistent state with respect to encryption, with no loss of user data.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

TABLE 1: EFS Encryption Process Summary

Step in Sequence	Process
1	The user profile loads to the Registry, if necessary.
2	EFS creates a log file named efsX.log in the System Volume Information subdirectory. X is a unique number in the filename (e.g., efs0.log). EFS writes to the log file when performing subsequent steps in the encryption process so that EFS can recover the file in case of system failure during the encryption process.
3	Microsoft Base Cryptographic Provider generates a random 128-bit FEK for the file.
4	EFS reads the HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash Registry value to identify the user's public key/private key pair.
5	EFS creates a DDF key ring with an entry for the user and associates the key ring with the file. The entry contains a copy of the FEK that the user's EFS public key encrypted.
6	EFS creates a DRF key ring for the file with an entry for each Recovery Agent on the system. Each entry contains a copy of the FEK that the Recovery Agent's EFS public key encrypted.
7	EFS creates a backup file, efsX.tmp, in the directory in which the file undergoing encryption resides. X is a unique number in the filename (e.g., efs0.tmp).
8	EFS places the DDF and DRF key rings in a header and adds the header to the file as the file's EFS attribute.
9	EFS marks the backup file as encrypted and copies the original file to the backup file.
10	EFS destroys the original file's contents and copies the backup to the original file. The copy operation results in the data's encryption, because the backup file is marked as encrypted.
11	EFS deletes the backup file.
12	EFS deletes the log file.
13	The user profile unloads from the Registry if it loaded in step 1.

As I stated in Part 1, the OS can designate directories as encrypted. EFS automatically encrypts any files a user moves into or creates in an encrypted directory.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

## The Decryption Process

The decryption process begins when a user opens an encrypted file. NTFS examines the file's attributes when opening the file, then executes the callback function `EfsOpenFile` in the EFS driver. `EfsOpenFile` reads the EFS attribute associated with the encrypted file. To read the attribute, `EfsOpenFile` calls several of the EFS support functions that NTFS exports for EFS's use. First, `EfsOpenFile` calls `NtOfsCreateAttributeEx` to open the attribute. `NtOfsCreateAttributeEx` supports opening any of a file's NTFS attributes, including the file's name attribute (which stores the file's name and attribute data), security attribute, and data attributes (which store the file data). For a refresher course on attributes, see NT Internals: "Inside NTFS," January 1998. To use `EfsOpenFile` to open the EFS attribute, EFS specifies `$EFS` as the attribute name and the numeric identifier of the `$EFS` attribute.

`EfsOpenFile` next opens the EFS attribute and calls the NTFS function `NtOfsQueryLength` to determine the attribute's length. Then, `EfsOpenFile` allocates a buffer to store the attribute's contents, calls `NtOfsMapAttribute` to obtain a virtual memory pointer to the contents, and copies the contents into the buffer. `EfsOpenFile` returns the EFS attribute data and bookkeeping information to NTFS. If `EfsOpenFile` can't open the EFS attribute, NTFS fails the file's open operation and returns an appropriate error code to the application that tried to open the file.

If the EFS attribute opens successfully, NTFS completes the necessary steps to open the file. When the file-open operation completes, NTFS invokes the EFS callback function `EFSFilePostCreate`. NTFS passes the context information from `EfsOpenFile` as a parameter to `EFSFilePostCreate`. `EFSFilePostCreate` ensures that the user opening the file has access to the file's encrypted data (i.e., that an encrypted FEK in either the DDF or DRF key rings corresponds to a public key/private key pair associated with the user). As EFS performs this validation, EFS obtains the file's decrypted FEK to use in subsequent data operations the user might perform on the file.

`EFSFilePostCreate` can't decrypt a FEK and relies on LSASRV (which can use the `CryptoAPI`) to perform FEK decryption. Before `EFSFilePostCreate` sends an LPC message to LSASRV, `EFSFilePostCreate` extracts the security ID (SID) of the user opening the file from the current process token. Then, `EFSFilePostCreate` attaches to the `lsass.exe` process (LSASRV's location). When a kernel thread attaches to a process, the thread associates with the process' (in this case `lsass.exe`) user-mode virtual memory; the thread doesn't attach to the virtual memory of the process to which the thread belongs (in this case the process opening the file). `EFSFilePostCreate` allocates a buffer in the LSASS process' virtual memory and copies the EFS context information—which includes the EFS attribute data that `EfsOpenFile` passed to `EFSFilePostCreate` indirectly—into the buffer. Because `EFSFilePostCreate` is now executing as a thread in the `lsass.exe` process (which executes in the System account) instead of as a thread in the process executing in the user's account that is opening the file, `EFSFilePostCreate` must impersonate the user opening the file. `EFSFilePostCreate` calls the `PsiImpersonateClient Security Manager` function to accomplish the impersonation. Now, all the information LSASRV needs is in place, and `EFSFilePostCreate` sends an LPC message by way of the `ksecdd.sys` driver to LSASRV. The LPC message asks LSASRV to obtain the decrypted form of the encrypted FEK in the EFS attribute data that corresponds to the user the thread is now impersonating.

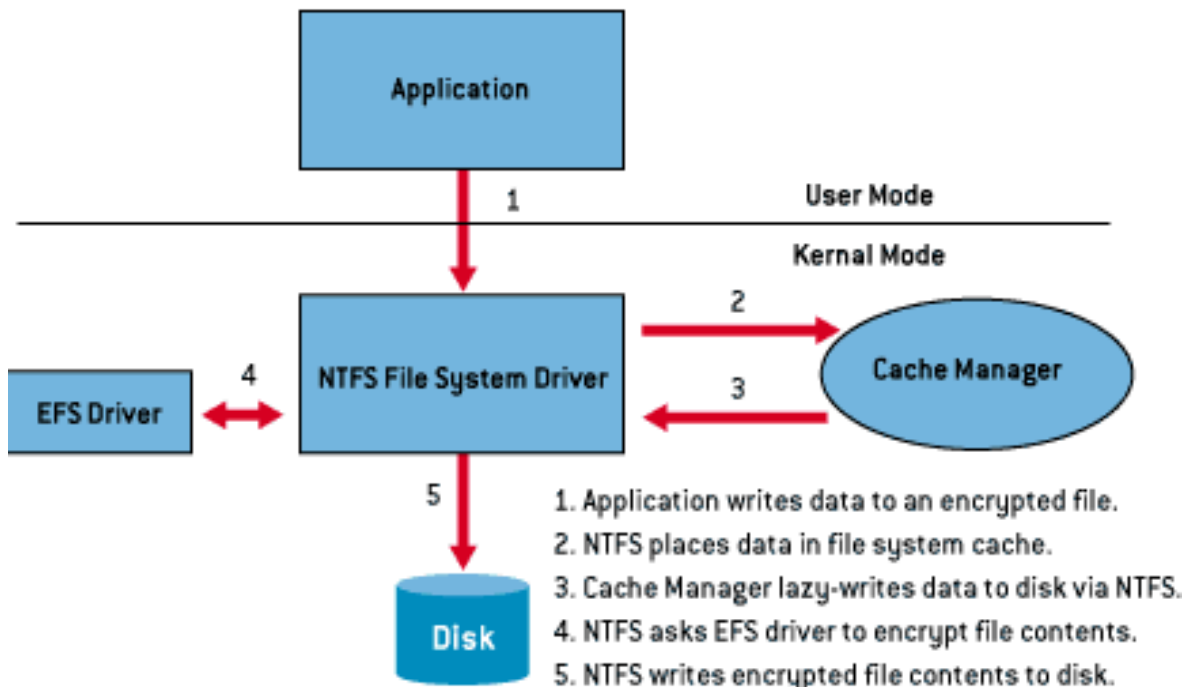
When LSASRV receives the LPC message, LSASRV executes the EFS function `LpcEfsDecryptFek`. To execute in the security context of the user opening the file, rather than in the System security context, `LpcEfsDecryptFek` uses an LPC impersonation facility that changes the function's security context to the user's security context. As I described in Part 1, both EFS and the `CryptoAPI` require access to a user's Registry profile settings; therefore, `LpcEfsDecryptFek` executes the `userenv.dll`

## Inside Encrypting File System

Mark Russinovich  
(Reprinted from WindowsItPro Magazine)

(User Environment DLL) LoadUserProfile API to bring the user's profile into the Registry, if the profile isn't already loaded.

LpcEfsDecryptFek calls the DecryptFek function to perform decryption. DecryptFek doesn't know which of the encrypted FEKs in the DDF and DRF key rings correspond to the user opening the file. To find the user's FEK, DecryptFek proceeds through each key field in the EFS attribute data, using the user's private key to try to decrypt each FEK. Figure 2, reprinted from Part 1, shows an example EFS attribute's format. For each key, DecryptFek calls the function GetFekFromEncryptedKeys, which attempts to decrypt a DDF or DRF key entry's FEK. GetFekFromEncryptedKeys first obtains a CryptoAPI reference to the user's private key. Last month, I explained that a certificate hash that the encryption process stores in a user's private key entry uniquely identifies a user's key pair. Because a user can have more than one EFS public key/private key pair, GetFekFromEncryptedKeys relies on the certificate hash in the key entry to identify the private key it will use to try to decrypt the encrypted FEK stored in that field. Because EFS stores EFS key pairs in the My certificate storage area, GetFekFromEncryptedKeys specifies the My storage area when requesting the CryptoAPI function CertOpenSystemStore to open the appropriate certificate store. GetFekFromEncryptedKeys gives CertFindCertificateInStore the current key entry's certificate hash, and CertFindCertificateInStore searches the My store for the key pair the certificate hash identifies.



If CertFindCertificateInStore doesn't find the key pair's certificate in the certificate store, it returns an error. In that case, GetFekFromEncryptedKeys moves on to the next key field. If DecryptFek can't decrypt any DDF or DRF key field's FEK, the user can't obtain the file's FEK. Consequently, EFS denies access to the application opening the file. However, if CertFindCertificateInStore is successful, DecryptFek uses CryptAcquireContext to open a cryptographic session with the security provider that issued the key pair the certificate hash designates. To open the session, CryptAcquireContext requires the name of the container storing the key pair, the name of the cryptographic provider, and the cryptographic provider's type. DecryptFek obtains this information by invoking the CertGetCertificateContextProperty CryptoAPI before calling CryptAcquireContext.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Although an EFS attribute's key fields store the key pair container name, provider name, and provider type, in Win2K beta 3, DecryptFek doesn't use the key fields' data. Win2K's release version EFS might use this information, however, to help it directly access a key pair by using CryptAcquireContext. Using the data stored in the key fields means EFS won't have to query the cryptographic provider for the container and provider name and provider type via CertGetCertificateContextProperty. This optimization can significantly speed EFS's access to encrypted files.

After DecryptFek opens a cryptographic session with the provider, DecryptFek accesses the user's private key via a call to CryptGetUserKey. DecryptFek never directly accesses the user's private key; rather, DecryptFek uses the call to CryptGetUserKey to tell the provider that DecryptFek wants to perform subsequent cryptographic functions with the user's key pair. Finally, DecryptFek invokes the CryptDecrypt CryptoAPI function to decrypt the FEK with the user's private key. As an extra security measure, DecryptFek calls the function EfspValidateEfsStream after CryptDecrypt decrypts the FEK. EfspValidateEfsStream obtains a Message Digest 5 (MD5) hash of the EFS attribute and decrypted FEK and compares the MD5 hash with the hash value that the EFS attribute's header stores. A mismatch between the hashes indicates that the EFS attribute is corrupt. In this case, EFS reads the current list of recovery agents, recomputes the checksum, applies the new EFS attribute to the file, then rebuilds the EFS attribute's DRF entries. By updating the DRF entries, EFS ensures that recovery agents can always access a file's data, even when the file's DDF entries are corrupted.

Because DecryptFek processes DDF and DRF key rings when decrypting a FEK, DecryptFek automatically performs file-recovery operations. If a recovery agent that isn't registered to access an encrypted file (i.e., it doesn't have a corresponding field in the DDF key ring) tries to access a file, EFS will let the recovery agent gain access because the agent has access to a key pair for a key field in the DRF key ring. EFS adds recovery-agent DRF entries to a DRF key ring for each recovery key pair in the system. A systems administrator can make any number of users recovery agents by assigning them access to an EFS recovery key pair.

## Keeping EFS Attribute Information Up-to-Date

EFS ensures that the information the EFS key fields store is always current. For example, users might receive new EFS certificates, or they might receive a new EFS key pair if their original pair is compromised. If the key field for which DecryptFek successfully decrypted the FEK is in the DDF key ring of the EFS attribute, DecryptFek calls UserKeyCurrent. If the field is in the DRF key ring, DecryptFek calls RecoveryInformationCurrent. These functions complete the same basic operation: They both compare the SID, cryptographic provider name, cryptographic container name, and cryptographic hash value stored in the EFS attribute with the user's SID and the properties of the user's current EFS cryptographic key pair. The

HKEY\_CURRENT\_USER\Software\Microsoft\WindowsNT\CurrentVersion\EFS\CurrentKeys\CertificateHash Registry key stores the user's active EFS key pair certificate hash. If any of the components in the key field don't match the current key pair's properties, including the user's SID, EFS must update the key field.

To update a key field in the DDF, DecryptFek calls ReplaceUserKey. To update a key field in the DRF, DecryptFek calls UpdateRecoveryInformation. These functions perform the same steps that ConstructKeyRing performs when a user encrypts a file for the first time. However, ReplaceUserKey and UpdateRecoveryInformation update the key fields instead of creating new fields. Both functions add the updated field to the end of the DDF or DRF key ring of the EFS attribute, then delete the old field.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

After DecryptFek successfully obtains the file's FEK, control returns to LpcEfsDecryptFek, which allocates a virtual memory buffer and copies the FEK into the buffer. If the user's profile loaded specifically for the FEK decryption, LpcEfsDecryptFek unloads the profile and returns the decrypted FEK to the EFS driver via an LPC reply message. The EFS driver's EFSFilePostCreate function copies the decrypted FEK from the buffer into a kernel buffer. Then, in a somewhat odd move, EFSFilePostCreate sends a control command to itself by way of the NTFS driver. This control command executes the SetEfsData function in the EFS driver, which takes the decrypted FEK and associates it with the opened file. The control command also updates the file's EFS attribute if EFS updated a DDF or DRF field. As with all control commands that the EFS driver interprets, EFS encrypts this command using DESX and the EFS session key before sending it.

## Decrypted FEK Caching

Traveling the path from EFS to LSASRV and back can take a relatively long time—in the process of decrypting a FEK, CryptoAPI uses results in more than 2000 Registry API calls and 400 file-system accesses on a typical system. The EFS driver, with the aid of NTFS, uses a cache to try to avoid this expense.

As I've described, NTFS keeps the EFS context information that the EFS driver associates with a file, and NTFS passes the context information to EFS when invoking an EFS callback function. NTFS stores EFS context information with NTFS's file context information, or file control block (FCB). After a user opens a file, NTFS optimizes subsequent opens of that file by keeping the FCB in an FCB table. By doing so, NTFS doesn't need to reconstruct the context information. When NTFS calls EFSFilePostCreate, NTFS passes the FCB to EFS so that EFS can see whether the FCB stores EFS's context information. If the FCB does store EFS's context information, EFS can choose to bypass the FEK decryption because EFS's context data stores the decrypted FEK.

EFS maintains a cache that stores user SID/EFS attribute-hash pairs. After a FEK is successfully decrypted, EFSFilePostCreate calls EfsRefreshCache to create a cache entry for the decrypted FEK. Thus, before EFSFilePostCreate goes to the trouble of requesting LSASRV to decrypt a file's FEK, EFSFilePostCreate looks in the FCB that NTFS handed to EFS to see whether EFS's context information is present. If the FCB contains the EFS context data, EFS has already decrypted the file's FEK. Therefore, EFSFilePostCreate calls EfsFindInCache to look for an entry in the EFS cache that contains the current user's SID (extracted from the token of the process opening the file) and the EFS attribute hash (taken from the FCB's EFS context information). If EfsFindInCache finds this entry, EfsFindInCache runs a check to determine whether the entry was created more than 5 seconds earlier. If the entry is older than 5 seconds, EFS discards the entry and follows the usual path through LSASRV to decrypt the FEK. If the entry is younger than 5 seconds, EFS uses the copy of the decrypted FEK in the EFS context data that the FCB has already associated with the file.

## Decrypting File Data

After an application opens an encrypted file, the application can read from and write to the file. NTFS calls the EFS callback EfsRead to decrypt file data as NTFS reads the data from the disk, and before NTFS places the data in the file system cache. Similarly, when an application writes data to a file, the data remains in unencrypted form in the file system cache until the application or the Cache Manager uses NTFS to flush the data back to disk. When an encrypted file's data writes back from the cache to the disk, NTFS calls the EFS callback EfsWrite to encrypt the data.

NTFS passes the file's FCB as a parameter to both EfsRead and EfsWrite. The FCB contains the file's EFS context data, and the EFS driver uses the file's decrypted FEK, which the EFS context data

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

includes, to perform DESX decryption or encryption of file data buffers. The EFS driver performs encryption and decryption in 512-byte units. The 512-byte size is the most convenient for the EFS driver, because disk reads and writes occur in multiples of the 512-byte sector.

## Backing Up Encrypted Files

An important aspect of any file encryption facility's design is that file data is never available in unencrypted form except to applications that access the file via the encryption facility. This restriction particularly affects backup utilities, in which archival media store files. EFS addresses this problem by providing a facility for backup utilities so that the utilities can back up and restore files in their encrypted states. Thus, backup utilities don't have to be able to decrypt file data, nor do the utilities decrypt file data in their backup procedures.

Backup utilities use the new EFS APIs `OpenEncryptedFileRaw`, `ReadEncryptedFileRaw`, and `WriteEncryptedFileRaw` in Win2K to access a file's encrypted contents. The `advapi32.dll` library provides these APIs, which all use LPC to invoke corresponding functions in `LSASRV`. For example, after a backup utility opens a file for raw access during a backup operation, the utility calls `ReadEncryptedFileRaw` to obtain the file data. The `LSASRV` function `EfsReadFileRaw` issues control commands (which the EFS session key encrypts with DESX) to the NTFS driver to first read the file's EFS attribute, then the encrypted contents.

`EfsReadFileRaw` might have to perform multiple read operations to read a large file. As `EfsReadFileRaw` reads each portion of such a file, `LSASRV` sends a remote procedure call (RPC) message to `advapi32.dll` that executes a callback function that the backup program specified when it issued the `ReadEncryptedFileRaw` API. `EfsReadFileRaw` hands the encrypted data it just read to the callback function, which can write the data to the backup media. Backup utilities restore encrypted files in a similar manner. The utilities call the `WriteEncryptedFileRaw` API, which invokes a call-back function in the backup program to obtain the unencrypted data from the backup media while `LSASRV`'s `EfsWriteFileRaw` function is restoring the file's contents.

## Viewing EFS Information

EFS has other APIs that applications can use to manipulate encrypted files. For example, applications use `AddUsersToEncryptedFile` to give additional users access to an encrypted file and `RemoveUsersFromEncryptedFile` to revoke users' access to an encrypted file. Applications use `QueryUsersOnEncryptedFile` to obtain information about a file's associated DDF and DRF key fields. `QueryUsersOnEncryptedFile` returns the SID, certificate hash value, and display information that each DDF and DRF key field contains. Screen 1 shows the output of `EFSDump`, a utility I developed that displays information that `QueryUsersOnEncryptedFile` returns. You can see that the file `mshtml.dll` has one DDF entry for user Joe and one DRF entry for Mark Russinovich, which is the only recovery agent currently registered on the system. You can download `EFSDump`, with full source code, from <http://www.sysinternals.com/misc.htm>.

# Inside Encrypting File System

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

```
Command Prompt

EFS Information Dumper v1.01
Copyright (C) 1999 Mark Russinovich
Systems Internals - http://www.sysinternals.com

mshtml.dll:
DDF Entry:
  NT5BETA2\Joe:
    CN=Joe,L=EFS,OU=EFS File Encryption Certificate
DRF Entry:
  NT5BETA2\MarkRussinovich:
    OU=EFS File Encryption Certificate, L=EFS, CN=MarkRussinovich

M:\temp>
```

## EFS Caveats

As you use EFS to protect your data, you need to be aware of some important concerns. First, compression and encryption aren't compatible. If you specify both attributes for a file, encryption will override compression.

A more important concern is that even after you encrypt an existing file, the file's original unencrypted data remains where NTFS stored the file on disk before you encrypted it. NTFS eventually reuses this disk space for storing other files and directories, but until NTFS reallocates the space, you can use a disk editor to bypass NTFS (e.g., from within NT or from a DOS boot disk) and directly access the unencrypted file data. The only way you can prevent such access is to run a utility that overwrites all free file system areas to destroy vulnerable information in the disk's unallocated portions. You can download one such utility—Sdelete—including its source code, from <http://www.sysinternals.com/sdelete.htm>.