# Global Information Assurance Certification Paper

# A Forensic Analysis of the Encrypting File System

*GIAC (GCFE) Gold Certification*

Author: Ramprasad Ramshankar, ramprasad.ramshankar@gmail.com
Advisor: Christopher Walker, CISSP, GSEC, GCED, CCISO, CISA
Accepted: January 27, 2021

Abstract

EFS or the Encrypting File System is a feature of the New Technology File System (NTFS). EFS provides the technology for a user to transparently encrypt and decrypt files. Since its introduction in Windows 2000, EFS has evolved over the years. Today, EFS is one of the building blocks of Windows Information Protection (WIP) - a feature that protects against data leakage in an enterprise environment (DulceMontemayor et al., 2019). From the attacker's perspective, since EFS provides out-of-the-box encryption capabilities, it can also be leveraged by ransomware. In January 2020, SafeBreach labs demonstrated that EFS could be successfully used by ransomware to encrypt files and avoid endpoint detection software (Klein A., 2020). The purpose of this paper is to provide security professionals with a better understanding of artifacts generated by EFS and recovery considerations for EFS encrypted files.

# 1. Introduction

In Windows 2000, Microsoft introduced built-in technologies that provided encryption of data in transit (IPSec) and data at rest (EFS) (Syngress, 2003). The EFS technology provides the capability for transparent encryption and decryption of files in an NTFS volume. Support for EFS has been built into NTFS starting from version 3.0 and above. The functionality provided by EFS differs from Bitlocker. Bitlocker, introduced in Windows Vista, provides volume level encryption capabilities whereas EFS provides encryption at the file content level.

EFS encryption is applicable to files as well as directories. When a directory is encrypted, all the files created inside that directory will automatically get encrypted. In the context of this paper, references to the term "files" also include directories. As a relevant aside, EFS and file-system compression are mutually exclusive. If a file is EFS encrypted then it cannot be compressed natively by NTFS and vice-versa.

To encrypt a file with EFS, Windows provides three options: the Windows GUI, the cipher command-line tool, and the Windows API. Through the Windows GUI, "users can encrypt files via Windows Explorer by opening a file's Properties dialog box, clicking Advanced, and then selecting the Encrypt Contents To Secure Data option." (Russinovich M. et al., 2012). The cipher is a built-in command-line tool that provides various options to create and manage EFS encrypted files (eross-msft et al., 2017). Notable are the /e and /d options that help to encrypt and decrypt files, respectively. The Windows API also provides a range of functions defined in Winbase.h (Winbase.h header, 2019) and Winefs.h (Winfesh.h, 2019) to programmatically work with EFS. The EFS service, hosted by the Local System Authority Sub-system (LSASS) process, transparently orchestrates all the related encryption/decryption operations.

EFS is also configurable through Group Policy. The MS-GPEF specification from Microsoft details the group policy settings that are available from enabling/disabling EFS to defining file recovery policies (Openspecs-office, 2018a)

Technically, EFS uses a combination of symmetric and asymmetric key cryptography to achieve file encryption/decryption. "Symmetric encryption algorithms are typically very fast, which makes them suitable for encrypting large amounts of data such as file data" (Russinovich M. et al.,2012). EFS, therefore, uses symmetric key algorithms (Openspecs-Office, 2018b) such as Advanced Encryption Standard (AES) and Triple Data Encryption Standard (3DES) to generate a per-file random key also called the File Encryption Key (FEK). Since the FEK is symmetric it can both encrypt and decrypt the contents of the file.

Anyone with access to a file's FEK will be able to decrypt the file contents, therefore the FEK must be protected such that only authorized user(s) will have access to it. EFS achieves FEK protection through the use of asymmetric key algorithms such as RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Key Cryptography). Asymmetric algorithms generate keys in pairs called the private key and the public key (Lastnaheholiu

et al., 2018b). When a piece of data is encrypted with a public key, it can only be decrypted with the corresponding private key and vice-versa.

EFS generates a public/private key-pair at the user level to protect the FEK. The user's public key is used to encrypt the FEK. "The source of the public key may be administratively specified to come from an assigned X.509 certificate or a smartcard or randomly generated" (Russinovich M. et al., 2012). This encrypted version of the FEK (EFEK) is stored as part of the encrypted file itself.

During decryption, the user's private key is essential to decrypt the EFEK. This private key is protected by a mechanism called Data Protection Application Programming Interface or DPAPI (Burzstein et al., 2010). The use of DPAPI ensures that the private key and the encrypted file are accessible only by an authorized user.

This paper explores the workings of EFS based on the outline provided above. The focus will be on-disk and in-memory artifacts generated by EFS.

## 2. Objective

The objective of this paper is to gain a practical understanding of EFS from a digital forensics perspective. There are three goals defined in this objective. The first goal is to study changes to an NTFS volume when a file gets encrypted. The second is to observe changes to a file's internal structures before and after encryption. The last goal is to put together this information to understand file recovery in EFS.

## 3. Methodology

The Update Sequence Number (USN) Journal is used to observe volume level changes during EFS encryption. The USN Journal is a per-volume file maintained by NTFS to record changes to files and directories in a volume (Mikeben et al., 2018a). The changes recorded in USN Journal will be in binary format and can be queried using the fsutil command-line tool (Toklima et al., 2018).

The Master File Table (MFT) is used to observe changes to a file's internal data structures during encryption. The MFT is one of the core files in NTFS. All the files in an NTFS volume have at least one entry in the MFT (Mikeben et al., 2018b), also called the file's MFT record. In exceptional cases a file can have multiple MFT records. These cases are outside the scope of this study. A comparison of a file's MFT record before and after encryption provides an understanding of EFS' workings.

This paper also makes references to registry keys used by EFS. These keys are identified using the Process Monitor tool from SysInternals (Russinovich M., 2020a). Process Monitor is used to monitor the registry changes made by lsass.exe – the

process that hosts the EFS service and mmc.exe – the process that writes group policy changes to the registry.

The tests related to this research are conducted in a Windows 10 standalone Virtual machine. The version of Windows used is 20H2 (determined by the *winver* command), and the version of NTFS is 3.1 (determined by running *fsutil fsinfo ntfsinfo c:*).

# 4. Volume Analysis

## 4.1. Approach

The approach taken for volume analysis is to review the USN journal changes when encrypting a plain text file. The file used for this example is "C:\Users\test\Documents\file.txt". This file is owned by the user "test" and is the first file in the "test" user's account to be encrypted with EFS.

Following is the summary of the volume level changes of encrypting a plain-text file. The detailed procedure is given in the author's blog (Diyinfosec, 2021a):

- A file named EFS0.LOG file is created in the "C:\System Volume Information" directory.
- The private and public key pairs are generated for the "test" user. The key pairs are stored in a file under the user profile inside the "%APPDATA%\Microsoft\Crypto\RSA\<SID>" directory.
- A certificate is created for the key pair in the "%APPDATA%\Microsoft\SystemCertificates\My\Certificates" of the user profile.
- A file named EFS0.TMP is created in the directory containing the plain text file- (C:\Users\test\Documents in our case). The EFS0.TMP acts as a placeholder for the plain text data as it gets encrypted. The EFS0.TMP is a hidden file owned by SYSTEM.
- The plain text file is marked as Encrypted.
- The encrypted content is written back to the plain text file (file.txt) effectively making it an encrypted file.
- EFS0.TMP file is cleared.
- EFS.LOG file is cleared.

The next section goes into the details of the above listed artifacts, their structure, and analysis.

## 4.2. Artifact Analysis

### 4.2.1. The EFS0.LOG file

EFS0.LOG is used to record the events during the encryption process (Syngress, 2003). This file is owned by SYSTEM and gets created under the "System Volume Information" directory of the volume containing the plain-text file. The EFS0.LOG gets removed after encryption. In the case of encrypting multiple files, the EFS0.LOG is created separately for each file (i.e., it does not get appended).

The structure of the EFS0.LOG is not documented by Microsoft. As part of this research, the log file was recovered using file recovery software (Active@ File Recovery, n.d.), and a few of the fields in the log were identified as documented in Appendix A.

EFS log records can be carved out from unallocated space in the volume. A Python script to carve the EFS log records is provided in author's Github page (diyinfosec, 2020d). During testing, between 1-3 copies of EFS log records were found per encrypted file in the unallocated space. EFS log records can provide evidence of a filename (with path) being present in an NTFS volume even if the file itself has been removed and its MFT record has been reused.

### 4.2.2. The Key-pair file

The EFS key-pair and its corresponding certificate (covered in the next section) are always generated together. Depending on the group policy settings, they can be generated on the domain controller or the local machine (Syngress, 2003). This research only covers locally generated key-pair files and self-signed certificates.

The key-pair/certificate is created at a user level, typically only once, i.e., when the user encrypts a file for the first time. Any files subsequently encrypted by that user can use the same public key to encrypt the FEK (which changes for every file) and the private key to decrypt the FEK.

EFS supports RSA and ECC algorithms for key-pair generation. The EFS Group policy settings determine the choice of algorithm and the length of the key (Openspecs-office, 2018c). The default algorithm used is RSA with a 2048-bit key.

The key-pair file is created under the "*%APPDATA%\Microsoft\Crypto\RSA\<SID>"* directory when RSA is used and under the "*%APPDATA%\Microsoft\Crypto\Keys*" directory when ECC is used.

Structurally the key-pair file has three elements - public key properties, private key properties, and an export flag. The public key properties are in clear text, whereas the private key properties and export flag are stored as DPAPI blobs. DPAPI ensures that only an authorized user will have access to the private key. The structure of the key-pair file is detailed further in Appendix B.

### 4.2.3. The Certificate file

The EFS certificate file is stored in the "%APPDATA%\Microsoft\SystemCertificates\My\Certificates" directory. This certificate acts as the binding between the user, the EFS key-pair, and the encrypted file. If the certificate file is deleted from the disk, the EFS encrypted files will no longer be accessible. The structure of the EFS certificate file is detailed in Appendix C.

A user can create a new EFS certificate and key-pair by using the cipher /k command. Any file subsequently encrypted by EFS will use the new certificate. If a user needs to use the new key-pair for already encrypted files, they can do so using the cipher /u command.

The thumbprint of the current EFS certificate is stored in the registry under HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys. This thumbprint can also be viewed using the cipher /Y command.

### 4.2.4. The EFS0.TMP file

Before a file is encrypted, its plain-text contents are copied onto a temporary file named EFS0.TMP. This is a hidden file owned by SYSTEM (Syngress, 2003) and is created in the same directory of the file being encrypted. The EFS0.TMP is removed once encryption completes.

When encrypting multiple files in a directory, the encryption proceeds sequentially, one file at a time. Therefore the EFS0.TMP is created and removed for each file getting encrypted. The only case where EFS0.TMP does not get created is when encrypting a zero-byte file.

An implication of the EFS0.TMP file is the additional space requirement when encrypting a file. For example, when a 500MB file is encrypted, the volume must have at least 500MB of free space. This free space allows EFS0.TMP to temporarily hold the contents of the plaintext file.

If a file getting encrypted has alternate data streams (ADS), then the same is reflected in the EFS0.TMP file as well. Figure 4-4 shows a file (010EditorWin64Installer100.exe) encrypted using the cipher command. This file is downloaded from the internet and therefore has an alternate data stream named Zone.Identifier (Openspecs-Office,2020a). The Zone.Identifier stream is also replicated in EFS0.TMP as seen from the output of the fls command in The Sleuth Kit toolset (Carrier B., n.d.)

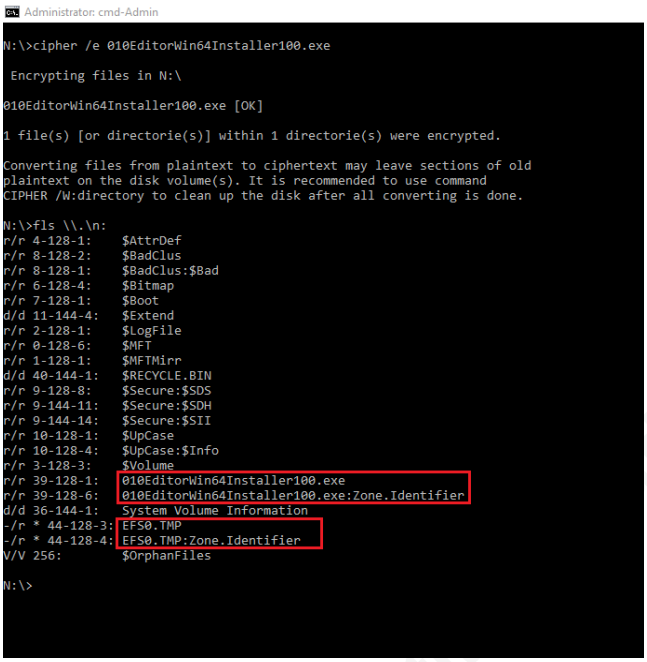


Figure 4-4: Illustrating EFS0.TMP creation with Alternate Data Streams

# 5. File Structure Analysis

## 5.1. Approach

File structure analysis involves studying a file's structure in the MFT before and after it is encrypted. To do this the mft2json tool is used (diyinfosec, 2020a). Mft2json is a Python-based tool created by the author. The tool reads a file's MFT record and prints it in JSON format. The JSON format allows easy inspection of changes to a file's MFT record. To ensure the accuracy of results, the findings are also verified by analyzing the MFT record with a disk editor (Active@ disk editor, n.d.)

A file in NTFS is a collection of attributes (NTFS File Types, n.d.). Each attribute holds a particular type of information about a file like timestamps, content, etc. An attribute structure has a header and a body. An attribute can either have a name or be unnamed. An MFT record is 1024 bytes in size. If an attribute can fit entirely inside a MFT record, this is called a resident attribute. If the contents of an attribute cannot fit inside a MFT record, there will be pointers from the MFT record referring to the clusters where the attribute information is stored. These attributes are called non-resident attributes.

The upcoming section will walk through attributes commonly found in an encrypted file and the changes to these attributes before and after encryption.

## 5.2. File Attribute Changes

### 5.2.1. The STANDARD_INFO attribute

Figure 5-1 shows a side-by-side comparison of the mft2json output of a file's STANDARD_INFO attribute before and after encryption. There are no changes to the attribute header during encryption and three changes inside the attribute body.

The first is the change of metadata time (si_ctime). This corresponds to the file encryption time. The second is the change to the access time (si_atime). The access time is in line with the encryption time. Though disabled in Windows 10 until version 19H2, access time updates appear to be on by default from Windows 10 version 20H1 (Suhanov, 2020). The third change is the file permissions (si_dos_perms) updated from "Normal" to "Archive|Encrypted".


```
{
  "attr_type":"0x10",
  "attr_type_str":"$STANDARD_INFORMATION",
  "attr_len":96,
  "attr_non_res_flg":0,
  "attr_name_len":0,
  "attr_name_offset":0,
  "attr_flags":"",
  "attr_id":0,
  "attr_body_len":72,
  "attr_body_offset":24,
  "attr_idx_flg":0,
  "attr_padding":0,
  "attr_name":"",
  "attr_body":{
    "si_btime":"29-Nov-2020 15:51:14",
    "si_mtime":"29-Nov-2020 15:51:14",
    "si_ctime":"29-Nov-2020 15:51:14",
    "si_atime":"29-Nov-2020 15:51:14",
    "si_dos_perms":"Normal",
    "si_max_versions":0,
    "si_version_num":0,
    "si_class_id":0,
    "si_owner_id":0,
    "si_security_id":5598,
    "si_quota_charged":0,
    "si_usn":0
  }
```

BEFORE ENCRYPTION

```
{
  "attr_type":"0x10",
  "attr_type_str":"$STANDARD_INFORMATION",
  "attr_len":96,
  "attr_non_res_flg":0,
  "attr_name_len":0,
  "attr_name_offset":0,
  "attr_flags":"",
  "attr_id":0,
  "attr_body_len":72,
  "attr_body_offset":24,
  "attr_idx_flg":0,
  "attr_padding":0,
  "attr_name":"",
  "attr_body":{
    "si_btime":"29-Nov-2020 15:51:14",
    "si_mtime":"29-Nov-2020 15:51:14",
    "si_ctime":"29-Nov-2020 15:51:16",
    "si_atime":"29-Nov-2020 15:51:16",
    "si_dos_perms":"Archive|Encrypted",
    "si_max_versions":0,
    "si_version_num":0,
    "si_class_id":0,
    "si_owner_id":0,
    "si_security_id":5598,
    "si_quota_charged":0,
    "si_usn":0
  }
```

AFTER ENCRYPTION


Figure 5-1: The MFT STANDARD_INFORMATION attribute of a file before/after encryption

### 5.2.2. The FILE_NAME attribute

There are no changes to the FILE_NAME attribute during file encryption. However, if a file is created inside an encrypted directory then the FILE_NAME attribute will have the same contents as the STANDARD_INFORMATION attribute.

### 5.2.3. The DATA attribute

The DATA attribute holds the contents of the file. It is the body of the DATA attribute that gets encrypted by EFS. If a file has multiple DATA attributes or Alternate Data Streams, all the data streams get encrypted.

Figure 5-2 shows the results of encrypting a resident DATA attribute. The data attribute header contains a flag (attr_flags) to indicate that its contents are encrypted (Russon R., n.d.).

Before encryption, this attribute was resident in the MFT. This can be inferred by the non-resident flag (attr_non_res_flg) being set to 0. Additionally, the file's content can be seen inside the attribute body (attr_body). The hex value "456E6372797074205468697321" corresponds to plain text "Encrypt This!".

After encryption the DATA attribute becomes non-resident. The attr_non_res_flg now has a value of 1 and attribute body (attr_body) contains an offset to encrypted content.

The DATA attribute of an encrypted file will always have the non-resident flag set to 1. This holds true even when encrypting a zero-byte file. However, a zero-byte file does not have any clusters allocated on the disk even though the data attribute has the resident flag set to 1.


Figure 5-2: The MFT DATA attribute of a resident file before/after encryption

Another noteworthy mention here is that EFS encryption happens at the cluster level. In this example, the plain text is 13 bytes long (attr_real_size). However, when looking at the cluster contents through Active@ Disk Editor, the entire cluster (4096 bytes in this case) was encrypted instead of just the 13 bytes of data.

Cluster-level encryption is an important factor to consider during file decryption/recovery. The Sleuth Kit tool icat only dumps the content of an attribute based on the real size (attr_real_size) and not the entire cluster. The author has raised an issue

in the SleuthKit project to fix this behavior, i.e., when icat dumps an encrypted attribute it should dump the last cluster entirely instead of rounding off to the real size found in the attribute header (diyinfosec,2020b)

### 5.2.4. The LOGGED_UTILITY_STREAM


Figure 5-3: The MFT LOGGED_UTILITY_STREAM attribute of a file, created after EFS encryption.

The LOGGED_UTILITY_STREAM is a new attribute that gets created and added to a file during encryption. As seen in Figure 5-3, this attribute is always non-resident (attr_non_res_flg is 1) and has the name "$EFS" (attr_name). The body of this attribute will be referred to below as "EFS Stream".

The EFS stream contains information to decrypt the file's DATA attribute(s). This information can be partially queried with the cipher /c <filename> command or by using the Efsdump tool from SysInternals (Russinovich M., 2006).

The EFS stream contains variable-sized records called Data Decryption Fields (DDF) or Data Recovery Fields (DRF). An EFS encrypted file will have at least one DDF record corresponding to the user that encrypted the file. If multiple users have

access to this file, then each user gets their own DDF record. A DRF record is created only if a recovery agent is configured for EFS. The structure of the EFS stream is detailed in Appendix D.

Inside a DDF record, there is the information required by EFS to decrypt the file. This includes, among other fields, the username, the user's public key thumbprint, and the EFEK. The DRF record contains similar information as a DDF but is intended for use by recovery agents.

During decryption, NTFS will need to parse the $EFS stream, extract the EFEK corresponding to the user, obtain the protected private key from the key-pair file, unprotect the private key using DPAPI, use the private key to decrypt the EFEK, and get the FEK. Finally, this FEK can be used to decrypt the contents of the DATA attribute(s).

## 5.3. Other Considerations

This section adds additional observations related to EFS encryption.

### 5.3.1. Directory Encryption

When a directory is marked as "encrypted" then all the files created in that directory are automatically encrypted. An encrypted directory also has a $EFS attribute and even a FEK. But the directory's FEK does not appear to serve any purpose. All the files inside the directory have their own unique FEKs.

### 5.3.2. Sparse files

When a sparse file is encrypted, the clusters marked as sparse are actually allocated on the disk and, then those clusters are encrypted. This results in a file effectively losing its sparseness. Interestingly, an encrypted sparse file is still marked as sparse (as queried using fsutil sparse queryFlag <filename>) even though all the clusters are allocated on disk during encryption.

# 6. File Recovery in EFS

The focus of this section is to outline recovery options for EFS encrypted files. Three types of recovery options are discussed here: Plain Text Recovery, Private Key Recovery, and FEK Recovery.

## 6.1. Plain Text Recovery

This section discusses approaches and considerations for the recovery of plaintext content of EFS encrypted files either in part or full.

### 6.1.1. Recovery using an EFS Recovery Agent

The simplest way to recover EFS encrypted files is to use the built-in agent-based recovery feature (Deland-Han et al., 2020a). If a recovery agent is configured, the FEK is also encrypted with the public key of the recovery agent and added to the $EFS stream. The private key of the recovery agent can be stored separately and used during recovery. The cipher /r command can be used to generate an EFS recovery key. The recovery key can be exported as a PFX archive (eross-msft, 2017) and added as a recovery agent via Group Policy.

### 6.1.2. Recovery from MFT Slack

The hypothesis here is that MFT record slack of a resident file might contain remnants of the plain-text contents after the file is encrypted. To test this hypothesis, a small plain-text file (468 bytes) was created, and then the file was encrypted using the cipher /e command. Figure 6-1 shows the results of the test. We can see that the MFT record slack space gets over-written with null bytes (0x00) during EFS encryption. Therefore, we can conclude that recovery of plain text from MFT slack is not possible.

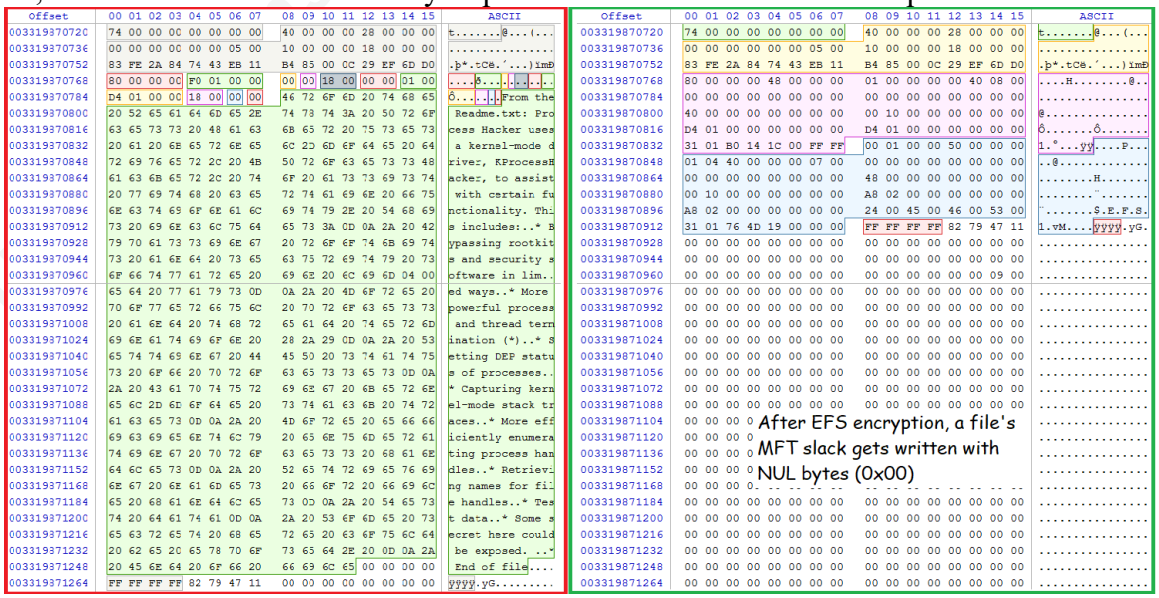


Figure 6-1: Comparing the MFT record of a resident file before/after encryption.

### 6.1.3. Recovery from Application temp files

There are file editing applications like Microsoft Word that create temporary copies of a file in the same directory before saving it (Microsoft Support,2020a). These applications might create a clear text temporary file when before saving an EFS encrypted file. This temporary file can be easily recovered using The Sleuth Kit or other

file recovery software. This problem does not exist if the directory containing the file is encrypted because then the temporary file is also encrypted by default.

This limitation is not a flaw in EFS but rather an exposure introduced by applications that are not EFS-aware. An example of an EFS aware application is Microsoft Office 2019 Professional. Word 2019 creates encrypted temporary files even if the directory itself is not marked as encrypted.

To avoid this type of file recovery, EFS provides a warning when encrypting only a file instead of encrypting the entire directory.


Figure 6-2: EFS warning about application behavior during encryption

### 6.1.4. Recovery from the EFS0.TMP file

Some of the unofficial NTFS documentation (Issues with EFS., n.d.) mentions that plain-text recovery is possible by recovering the EFS0.TMP file from the unallocated space. However, in NTFS version 3.1, the EFS0.TMP is always fully over-written with null bytes (0x00). Therefore plain-text recovery using EFS0.TMP is not possible in NTFS version 3.1.

### 6.1.5. Recovery from NTFS $LogFile

The NTFS journaling file or the $LogFile is used to recover the file system during a system crash or power failure (Cho G., Rogers M., 2012). The log file contains multiple entries or log records. A log record contains, among other fields, an operation code, an undo entry and a redo entry.

When a MFT resident file is encrypted, there is an entry created in the log file for the DeleteAttribute Operation (Operation Code: 0x06). This entry contains the plaintext contents of the file. After encryption, the original plaintext contents can be recovered by searching the $LogFile for the DeleteAttribute operation (0x06) and the DATA attribute type (0x80).

There are a few caveats to this approach. Firstly, this approach only works for resident plain-text files that are subsequently encrypted. Secondly, the $LogFile entries can get overwritten during normal file system operation. Additionally, running the cipher /w command to cleanup unallocated space (Deland-Han et al., 2020a) can also hasten the overwriting of data in the $LogFile.

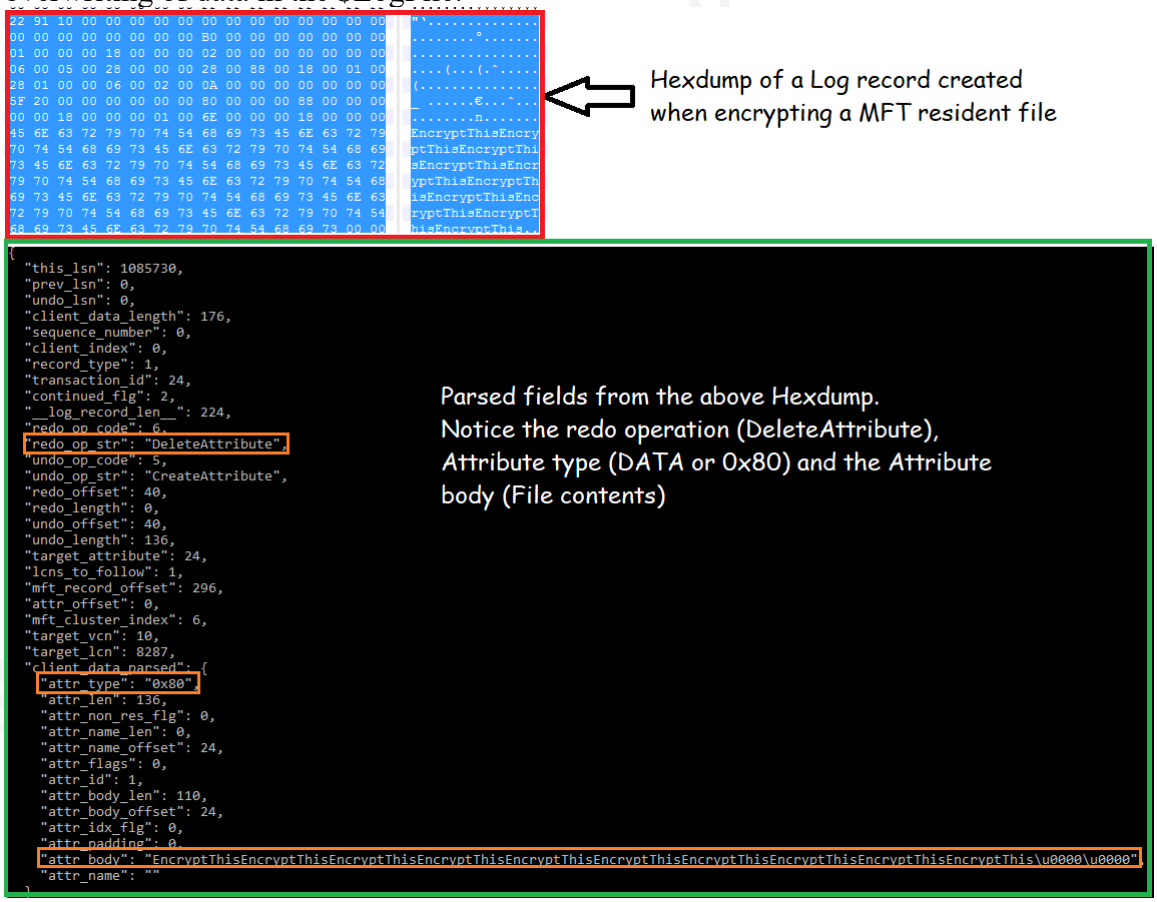


Figure 6-3 – The NTFS $LogFile showing contents of a MFT resident file.

### 6.1.6. Hypothesizing with file metadata

EFS encrypts only the DATA attribute of a file. Other identifying information about the file, such as the filename, timestamps, and size, is still in cleartext. Even if content recovery is not possible, contextual inferences can be made from the file metadata.

## 6.2. Private Key Recovery

The EFS private key does not directly provide access to an EFS encrypted file, but it can be used to decrypt the EFEK, obtain the FEK, and use the FEK to decrypt the file.

### 6.2.1. Extracting private key from the DPAPI blob

As discussed in the section "The Key-pair file", the EFS private key is stored as a DPAPI blob in a file inside the user's APPDATA directory. The full workings of DPAPI are outside the scope of this paper. However, the relevant parts are explained below.

A DPAPI blob is encrypted with a symmetric key known as Master Key (Grafnetter M., 2020). The Master Key related to a user is stored as a file in the C:\%APPDATA%\Microsoft\Protect\<SID>\ directory. There can be multiple Master Keys for the user in this directory (Picasso F., 2014).

The Master Keys in turn are protected by a pre-key derived from the user's password. If the user's password can be obtained through either social engineering/brute-forcing etc., then the password can be used to generate the pre-key and the pre-key can be used to decrypt the Master Key file. The mimikatz tool (Gentilkiwi, n.d.) can perform offline decryption of the DPAPI blob and obtain the EFS private key. The overview of the steps will be:

1. **Extract all the available Master Keys**

   The below mimikatz command needs to be run for every Master Key file available in the %APPDATA%\Microsoft\Protect directory. The output of this command (given the correct user password) will be a Master Key.

   *dpapi::masterkey*
   */in:"C:\%APPDATA%\Microsoft\Protect\<SID>\<filename>"*
   */password:<user_password>*

   Alternately, the lsass.exe memory also holds a copy of all the Master Keys. These can be carved from lsass.exe using the mimikatz command "privilege::debug sekurlsa::dpapi dapapi::cache". For domain-joined users, the Domain Backup Key from Active Directory can also decrypt the Master Key (Grafnetter M., 2015).

2. **Use the Master Key to decrypt the private key DPAPI blob**

   The EFS private key files are found in either the "%APPDATA%\Microsoft\Crypto\RSA\<SID>" or the "%APPDATA%\Microsoft\Crypto\Keys" directories. To obtain the private key with the Master Key the following mimikatz command can be used

(Delpy B., 2017). Once the Master Keys are obtained, they can be iteratively tried to decrypt the EFS private key using the below mimikatz command.
*dpapi::capi /in:"C:\path\to\private\key\file " /masterkey:<masterkey>*

### 6.2.2. Extracting private key from the EFSUI process

efsui.exe is a process started under lsass.exe that generates a prompt for the user to back up their EFS certificate. The efsui.exe allows a user to export their EFS certificate and the corresponding private key as a Publisher Information Exchange (PFX) file. Theoretically, the EFS private key should be recoverable by carving the memory of the efsui.exe process. Further research is needed to establish how the private key is stored inside the efsui.exe process memory.

### 6.2.3. Next Steps after obtaining the private key

Obtain the EFEK by parsing the $EFS attribute of the encrypted file (Appendix B-4). Decrypt the FEK using the private key. Use the FEK to decrypt the file contents. A walkthrough of decrypting an EFS encrypted file is available on the author's Github page (diyinfosec, 2020c).

## 6.3. FEK Recovery

Obtaining the FEK allows us to directly decrypt an EFS encrypted file. This research identifies two areas in memory where the FEKs can be found. The first is the kernel memory, and the second is the process memory of lsass.exe.

### 6.3.1. FEK recovery from Kernel Memory

The approach to identify FEK in kernel memory is documented in the author's blog (Diyinfosec,2021b), and only the results are mentioned here. FEKs can be found in the non-paged pool region of the kernel memory. The kernel data structure containing the FEK is identifiable by the pool tag "Efsm".

EFS FEKs are recoverable by scanning the memory for Efsm pool tags. These keys are written to Efsm pool objects during EFS encryption and will be lost when the host is shutdown/rebooted.

### 6.3.2. FEK recovery from lsass.exe

When an encrypted file is opened, it's FEK is also loaded into the memory of the lsass.exe process. The in-memory structure of the FEK is documented in the EFSRPC protocol specifications (Openspecs-office, 2018e). It should be straightforward to carve FEKs from lsass.exe memory based on this structure. The cipher /flushcache command clears the FEK from lsass.exe memory if the corresponding file is no longer open.

### 6.3.3. Next Steps after obtaining the FEK

The challenge with obtaining the FEK from memory is that there is no clear attribution between the FEK and the encrypted file. So the FEKs got using this approach will have to be iteratively tried over all the EFS encrypted files in that host to check for successful decryption. Once the FEK is obtained it can be used to decrypt a file. A walkthrough of decrypting an EFS encrypted file is available on the author's Github page (diyinfosec, 2020c).

## 7. Conclusions

EFS is a robust system that provides transparent encryption/decryption of files in an NTFS volume. EFS relies on the security of the user's password to protect the private key, which in turn protects the File Encryption Key.

This research has documented the structure and purpose of the artifacts generated during EFS encryption from a digital forensics perspective. This has yielded a new approach to identify the file name and path of deleted EFS encrypted files. Additionally, a couple of techniques to identify EFS file encryption keys in memory have been outlined.

There is potential for further research in EFS, especially around key recovery from memory and the forensic artifacts generated by EFS when used with Windows Information Protection (WIP).

## 8. References

Active@ disk editor. (n.d.). Active@ Disk Editor - Freeware Hex Viewer & Hex Editor for raw sectors/clusters. https://www.disk-editor.org/index.html

Active@ File Recovery (n.d.). LSoft Technologies - Data Security, Data Backup, Data Recovery, Industrial Solutions. https://www.lsoft.net/file-recovery/

Bruno (2014, April 30) How to get SSL-certificate sha1 fingerprint? Stack Overflow. https://stackoverflow.com/questions/23371619/how-to-get-ssl-certificate-sha1-fingerprint

Burzstein E., Picod J. (2010). Recovering Windows Secrets and EFS Certificates Offline https://www.usenix.org/legacy/event/woot10/tech/full_papers/Burzstein.pdf

Carrier B. (n.d.) The sleuth kit. https://www.sleuthkit.org/sleuthkit/

Cho G., Rogers M. (2012, January) Finding forensic information on creating a folder in $LogFile of NTFS. ResearchGate.

Author Name, email@address

https://www.researchgate.net/publication/288987157_Finding_Forensic_Information_on_Creating_a_Folder_in_LogFile_of_NTFS

Colridge R. (1996, August 19). The Cryptography API, or How to Keep a Secret. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/previous-versions/ms867086(v=msdn.10)

Deland-Han, xl989, simonxjx (2020, September 8). Back up recovery agent EFS private key - Windows Server. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/troubleshoot/windows-server/identity/back-up-recovery-agent-efs-private-key

Deland-Han, xl989, simonxjx (2020b, September 8). Use Cipher.exe to overwrite deleted data - Windows Server. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/troubleshoot/windows-server/windows-security/use-cipher-to-overwrite-deleted-data

Delpy B. (2017, October 8). GitHub. https://github.com/gentilkiwi/mimikatz/wiki/howto-~-decrypt-EFS-files

diyinfosec (2019, December 23). diyinfosec/010-Editor. GitHub. https://github.com/diyinfosec/010-Editor/blob/master/WINDOWS_CERTIFICATE_BLOB.bt

diyinfosec (2020a, October 29). diyinfosec/mft2json. GitHub. https://github.com/diyinfosec/mft2json

diyinfosec (2020b, January 26). [NTFS - Encrypted Data attribute] icat should dump entire cluster instead of using the init_size/real_size · Issue #1798 · sleuthkit/sleuthkit. GitHub. https://github.com/sleuthkit/sleuthkit/issues/1798

diyinfosec (2020c, March 27). YT_Exercises. GitHub. https://github.com/diyinfosec/YT_Exercises/tree/master/NTFS_EFS_Decryption

diyinfosec (2020d, Oct 25). diyinfosec/efs_test. GitHub. https://github.com/diyinfosec/efs_test/blob/main/efslog_extract.py

Diyinfosec. (2021a, January 27). Monitoring USN journal for changes. Medium. https://diyinfosec.medium.com/monitoring-usn-journal-for-changes-e72069b18441

Diyinfosec. (2021b, January 27). Scanning memory for FEK. Medium. https://diyinfosec.medium.com/scanning-memory-for-fek-e17ca3db09c9

Author Name, email@address

DulceMontemayor, tiburd, VLG17, LauraKellerGitHub, garycentric, martyav, Dansimp, imba-tjd, get-itips, Justinha, coreyp-at-msft, cloudhandler (2019, March 05). Protect your enterprise data using Windows information protection (WIP). Technical documentation, API, and code examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/security/information-protection/windows-information-protection/protect-enterprise-data-using-wip

eross-msft, john-par, mijacobs, coreyp-at-msft, b-r-o-c-k, lizap (2017, October 16). Cipher. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cipher

Gentilkiwi (n.d.). mimikatz. GitHub. https://github.com/gentilkiwi/mimikatz

Google (n.d.). rekall. GitHub. https://github.com/google/rekall

Grafnetter M. (2020, March 24) #CQLabs - Extracting roamed private keys from active directory. CQURE Academy. https://cqureacademy.com/blog/extracting-roamed-private-keys

Grafnetter M. (2015, October 26) Retrieving DPAPI backup keys from active directory – Directory services internals. (n.d.). Directory Services Internals. https://www.dsinternals.com/en/retrieving-dpapi-backup-keys-from-active-directory/

Hao X. (2009, May 11) Attack Certificate-based Authentication System and Microsoft InfoCard https://powerofcommunity.net/poc2009/xu.pdf

Issues with EFS. (n.d.). NTFS.com - Data Recovery Software, File Systems, Hard Disk Internals, Disk Utilities. https://ntfs.com/issues-encrypted-files.htm

Ireland D. (2020, May 12). Padding schemes for block ciphers. CryptoSys cryptography software tools for Visual Basic and C/C++/C# developers. https://www.cryptosys.net/pki/manpki/pki_paddingschemes.html

juanlang et al. (2020, March 17). wine-mirror/wine. GitHub. https://github.com/wine-mirror/wine/blob/master/include/wincrypt.h#L2426

Kexugit. (n.d.). Designing and implementing a PKI: Part III certificate templates. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/archive/blogs/askds/designing-and-implementing-a-pki-part-iii-certificate-templates

NTFS File Types (n.d.) NTFS.com - Data Recovery Software, File Systems, Hard Disk Internals, Disk Utilities. http://ntfs.com/ntfs-files-types.htm

Author Name, email@address

Karl-bridge-microsoft. (2018, December 05). Filetime (minwinbase.h) - Win32 apps. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-filetime

Klein A. (2020, January 21). EFS ransomware threat | SafeBreach labs. Breach and Attack Simulation. https://safebreach.com/Post/EFS-Ransomware

Layout.h. (n.d.). Open Source - Releases. https://opensource.apple.com/source/ntfs/ntfs-91.20.2/newfs/layout.h.auto.html

Lastnameholiu. (2018a, December 5). RSAPUBKEY structure (wincrypt.h) - Win32 apps. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/ns-wincrypt-rsapubkey

Lastnameholiu, drewbatgit, DCtheGeek, mijacobs, msatranjr (2018b, May 31) Asymmetric Keys. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/seccrypto/public-private-key-pairs

Lastnameholiu, DCtheGeek, mijacobs, msatranjr (2018c, May 31). CNG DPAPI. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/seccng/cng-dpapi

Microsoft Support (2020a, November 20). Description of how Word creates temporary files https://support.microsoft.com/en-ie/help/211632/description-of-how-word-creates-temporary-files

Mikben, msatranjr (2018a, May 31). Change Journals. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-gb/windows/win32/fileio/change-journals

Mikben, DCtheGeek, drewbatgit, msatranjr (2018b, May 31). Master File Table (Local File Systems). Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/fileio/master-file-table

Openspecs-office. (2018a, September 12). [MS-GPEF]: Group policy: Encrypting file system extension. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gpef/14d3fd83-7537-41a2-af39-8e52c19ef0e3

Openspecs-office. (2018b, September 12). [MS-EFSR]: ALG_ID. Developer tools, technical documentation and coding examples | Microsoft

Author Name, email@address

Docs. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-efsr/4cc8e6ae-99aa-483e-99c1-ab4c0ee46294

Openspecs-office. (2018c, September 12). [MS-GPEF]: EFS additional options. Developer tools, technical documentation and coding examples | Microsoft Docs https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gpef/cc14847c-7ba5-4c34-a12c-f167bdc54d1e

Openspecs-office. (2018d, September 12). [MS-EFSR]: Glossary. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-efsr/230807ac-20be-494f-86e3-4c8ac23ea584

Openspecs-office. (2018e, September 12). [MS-EFSR]: Encrypted FEK. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-efsr/00933615-c9cf-4d51-9d9a-bb3fb33a3560

Openspecs-office. (2020a, August 26). [MS-FSCC]: Zone.Identifier Stream Name. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/6e3f7352-d11c-4d76-8c39-2516a9df36e8

Picasso F. (2014), give me the password and I'll rule the world https://digital-forensics.sans.org/summit-archives/dfirprague14/Give_Me_the_Password_and_Ill_Rule_the_World_Francesco_Picasso.pdf

Russinovich M. (2006, November 01). EFSDump - Windows Sysinternals. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/sysinternals/downloads/efsdump

Russinovich M., Solomon D, Ionsecu A. (2012). Windows Internals.6th Edition, Part 2. Redmond, WA: Microsoft Press.

Russinovich M (2020a, September 17). Process Monitor - Windows Sysinternals. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

Russon R. (n.d.). Attribute - Concept - NTFS Documentation. FlatCap. https://flatcap.org/linux-ntfs/ntfs/concepts/attribute_header.html

Suhanov M. (2020 June 03). OneDrive and NTFS last access timestamps. My DFIR Blog. https://dfir.ru/2020/05/23/onedrive-and-ntfs-last-access-timestamps/

Author Name, email@address

Syngress (2003, February 28). MCSE/MCSA Implementing and Administering Security in a Windows 2000 Network (Exam 70-214) - 1st Edition.

Toklima, eross-msft, lizap, coreyp-at-msft (2018b, August 21). fsutil. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/fsutil

Winefs.h header (2019, January 11). Win32 apps. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/api/winefs/

Winbase.h header (2019, January 11). Win32 apps. Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/api/winbase/

# 9. Appendix

## 9.1. Appendix A – Structure of the EFS0.LOG file

The EFS0.LOG file consists of a single EFS log record. The log record is a variable length data structure that begins with the signature G.U.J.R (Unicode) or '4700 5500 4A00 5200' (Hex). Figure B1-1 shows the fields identified inside a log record.

| Byte Range | Structure | Name | Validity | Notes |
|---|---|---|---|---|
| 0-7 | char | Signature | Valid | G.U.J.R (Unicode) |
| 8-11 | uint32 | Unknown 1 | Unknown | Observed values: 100 |
| 12-15 | uint32 | encryption_chunk_size | Questionable | Observed values: 512 |
| 16-19 | uint32 | record_len | Valid | Length of the EFS Log record |
| 20-23 | uint32 | Unknown 2 | Unknown | Observed values: 512 |
| 24-27 | uint32 | Unknown 3 | Unknown | Observed values: 0 |
| 28-31 | uint32 | offset_to_src_file_name | Valid | File to encrypt – name with path |
| 32-35 | uint32 | src_file_name_len | Valid | Length of file to encrypt with path |
| 36-39 | uint32 | offset_to_dst_file_name | Valid | Offset to temp file (EFS0.TMP) name with path |
| 40-43 | uint32 | dst_file_name_len | Valid | Length of temp file with path |
| 44-47 | uint32 | Unknown 4 | Questionable | Observed values: 512 |
| 48-51 | uint32 | mft_record_size | Questionable | Observed values: 1024 |
| 52-55 | uint32 | Unknown 5 | Unknown | Observed values: 0 |
| 56-59 | uint32 | Unknown 6 | Unknown | Observed values: 0 |
| 60-63 | uint32 | Unknown 7 | Unknown | Observed values: 0 |
| 64-67 | uint32 | Unknown 8 | Unknown | Observed values: 8 |
| 68-71 | uint32 | Unknown 9 | Unknown | Observed values: 0 |
| 72-75 | uint32 | mft_id_encrypted_file | Valid | MFT record number of the file to encrypt |
| 76-79 | uint32 | Unknown 10 | Unknown | Observed values: 65536 |
| 80-83 | uint32 | Unknown 11 | Unknown | Observed values: 8 |
| 84-87 | uint32 | Unknown 12 | Unknown | Observed values: 0 |
| 88-91 | uint32 | mft_id_efs_tmp_file | Valid | MFT record number of temp file (EFS0.TMP) |
| 92-93 | uint16 | Unknown 13 | Unknown | Observed values: 0 |
| 94-95 | uint16 | Unknown 14 | Unknown | Indeterminable |
| 96-Variable | char | src_file_name | Valid | File to encrypt with path. |
| Variable | char | dst_file_name | Valid | Temporary file with path. |

Figure A-1: Fields in the EFS0.LOG file

## 9.2. Appendix B – Structure of the EFS key-pair

The EFS key-pair files will be in binary format and can be parsed using mimikatz. The structure of the key-pair file is different when using RSA and ECC, respectively. Both formats are discussed below.

With RSA, the key-pair file is in the legacy Crypto API or CAPI format. This file is created in the "*%APPDATA%\Microsoft\Crypto\RSA\<SID>*" directory and can be parsed by mimikatz using the dpapi::capi module. CAPI here refers to the legacy Microsoft crypto API introduced in Windows NT 4.0 (Colridge R.,1996)

Figure B-1 shows the data structures obtained from running the mimikatz command, *dpapi::capi /in:<key-pair-file-name>*. We can see that the key-pair file starts with a Crypto API header and a unique provider name value. This unique provider name provides the link between the key-pair file and the certificate file.

Following the unique provider name is the RSA public key. This RSA public key will be in clear text and will follow the RSAPUBKEY struct format defined in wincrypt.h (Lastnameholiu, 2018a). After the public key, there are two DPAPI protected blobs. The first blob contains an encrypted version of the RSA private key and the second blob

contains an encrypted Export flag. The RSA private key is used to decrypt the EFEK, typically when the user opens an EFS encrypted file. The Export flag contains one of the values defined under NCRYPT_EXPORT_POLICY_PROPERTY in ncrypt.h (Hao X, 2009).

With ECC, the location and structure of the key-pair are different than when using RSA. The key-pair file is created under a users' "*%APPDATA%\Microsoft\Crypto\Keys*" directory.

The ECC key-pair file is in binary format and can be parsed with mimikatz using the dpapi::cng module. CNG here refers to Cryptography Next Generation API, introduced in Windows Vista as a long-term replacement for the legacy Crypto API (Lastnameholiu, 2018c).

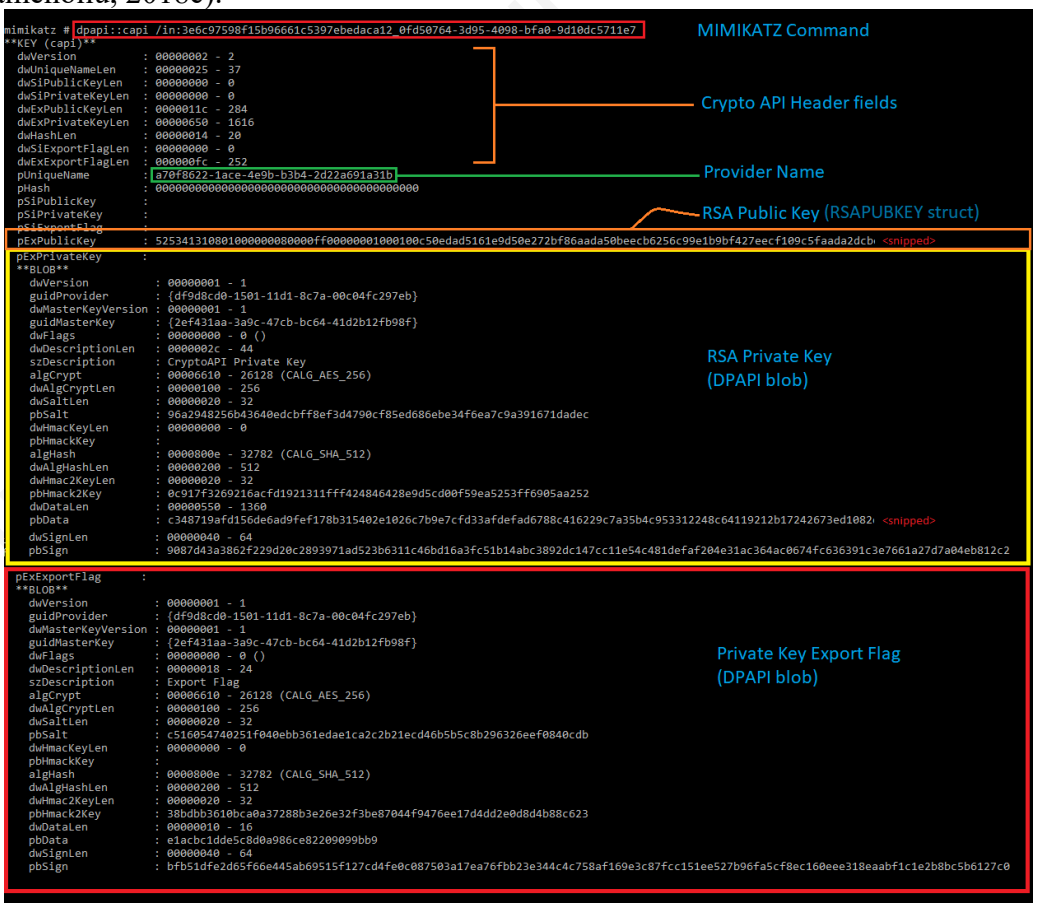


Figure B-1: Structure of the key-pair file when RSA algorithm is used.

Figure B-2 shows the data structures obtained from running the mimikatz command, *dpapi::cng /in:<key-pair-file-name>*. The key-pair file starts with a CNG API header and a unique provider name followed by a few ECC Public key property structures. The first public key property indicates the last modified time of the key-pair file in FILETIME format (Karl-bridge-microsoft, 2018). The next property is the ECC public key and the last one is the ECC certificate encoded using Distinguished Encoding Rules (DER) algorithm. The public key properties are in clear text. After the public key

properties, there are two DPAPI blobs, similar to the RSA key-pair file. The first blob contains an encrypted version of the key export flag, and the second blob contains the encrypted ECC private key.

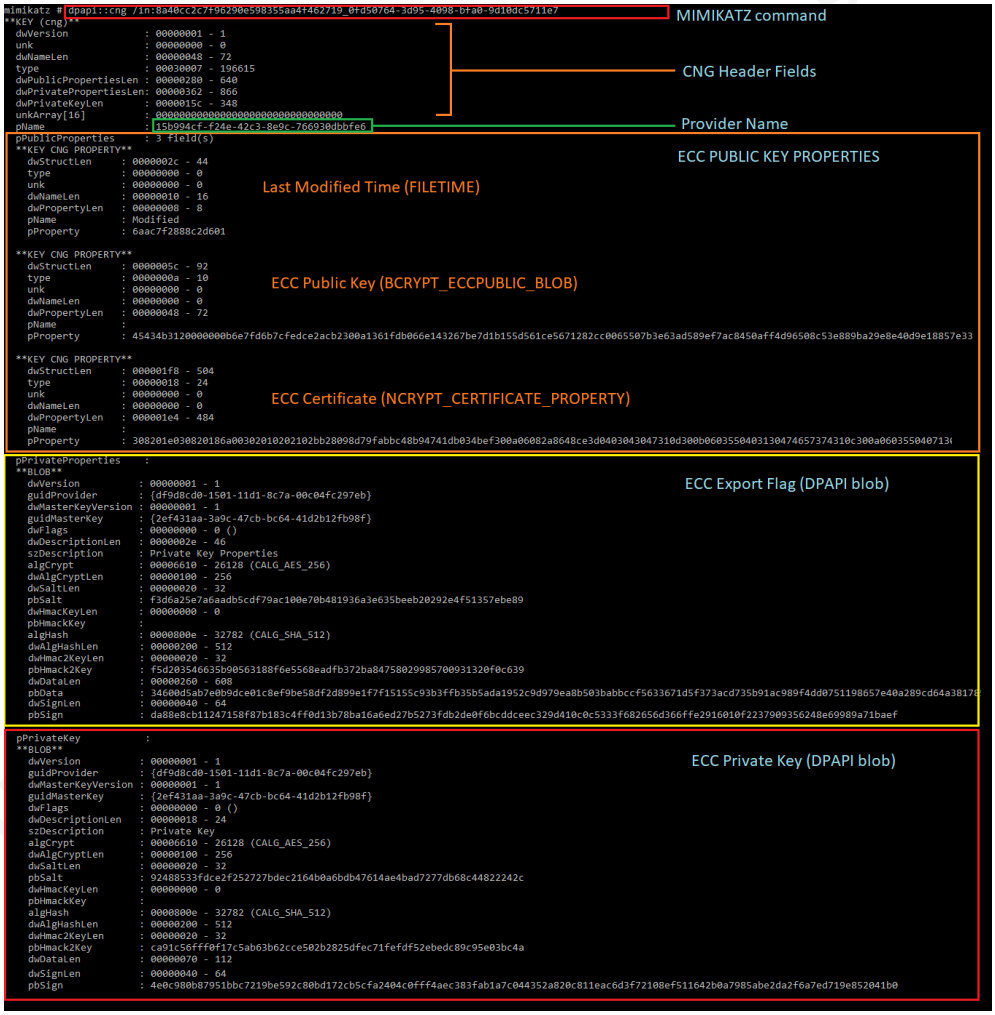


Figure B-2: Structue of the key-pair file when ECC algorithm is used.

## 9.3. Appendix C – Structure of EFS certificate

"A certificate is a collection of attributes and extensions that can be stored persistently. The set of attributes in a certificate can vary depending on the intended usage of the certificate." (Openspecs-office, 2018d)

As mentioned in "The Certificate file" section, the certificate serves as a link between the key-pair file and the encrypted file. The connection between the certificate and the key-pair is the unique provider name value. This value is found in both the key-pair file and also in the EFS certificate file as the CERT_KEY_PROV_INFO_PROP_ID attribute (Hao X, 2009). The certificate file also contains information about the user for

whom the certificate is issued and the certificate thumbprint. The certificate thumbprint is the SHA1 hash of the DER-encoded version of the certificate (Bruno, 2014). Lastly, the $EFS attribute contains the certificate thumbprint and the username info, linking it to the certificate.

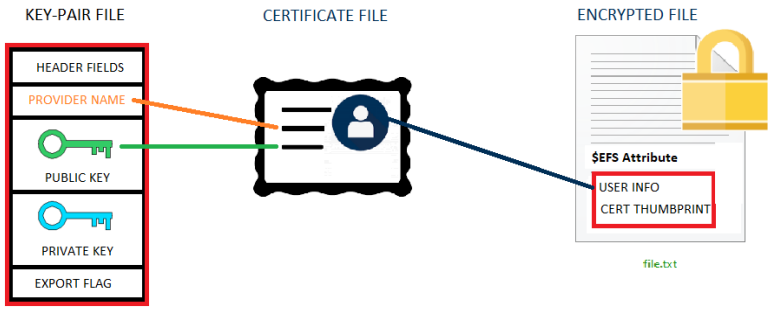


Figure C-1: Relationship between key-pair file, certificate file and the encrypted file

The attributes of the EFS certificate can be defined at the Enterprise Certificate Authority level using Certificate Templates (Kexugit, n.d.). If this is not set up or if the computer is not a part of a domain-joined environment, EFS will create a self-signed certificate. The self-signed certificate is valid for one hundred years by default.

Structurally, the certificate file contains a repeating array of Type-Length-Value (TLV) entries in the format defined in Figure B3-2. The Property ID is an integer that maps to the property definitions of CertSetCertificateContextProperty in wincrypt.h. The Property Data is the value of the property.

| Structure | Name | Validity | Notes |
|---|---|---|---|
| uint32 | Property ID | Valid | Defined in wincrypt.h |
| uint32 | Unknown 1 | Unknown | Observed values: 1 |
| uint32 | Property Size | Valid | Size of Property Data |
| variable | Property Data | Valid | The actual value |

Figure C-2: The structure used to store certificate attributes on-disk

The author has also created a 010 Editor template that can be used to parse a certificate file (diyinfosec, 2019). The property ID to property name mapping was obtained from the wincrypt.h implementation in WINE (juanlang et al., 2020).

## 9.4. Appendix D – Structure of the $EFS attribute

In terms of structure, the EFS stream contains a stream header followed by a number of variable sized records. These records are called Data Decryption Fields (DDF) or Data Recovery Fields (DRF). Each user having access to the encrypted file will have a DDF record entry. Likewise, each recovery agent will have a DRF entry. The structure of a DRF record is the same as the DDF record, but instead of user information, it will have

information about the recovery agent.

The structure of the EFS stream is given in Figure B4-1. This information was obtained from the "layout.h" file in Apple's open-source implementations of NTFS (Layout.h, n.d.). If the user or a recovery agent uses an ECC certificate, then the structure of the EFS stream changes. This new structure is documented in Figure B4-2.

| EFS Stream Header | | | | |
|---|---|---|---|---|
| **Byte Range** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-3 | uint32 | attribute_size | Valid | Size of the entire attribute in bytes |
| 4-7 | uint32 | unknown1 | Unknown | Unidentified |
| 8-11 | uint32 | efsVersion | Questionable | Observed values: 4 |
| 11-15 | uint32 | cryptoAPIVersion | Questionable | Observed values: 0 |
| 16-31 | byte | unknown_hash_1 | Questionable | Possibly MD5 of some struct. |
| 32-47 | byte | unknown_hash_2 | Valid | Possibly MD5 of some struct. |
| 48-63 | byte | unknown_hash_3 | Valid | Possibly MD5 of some struct. |
| 64-67 | uint32 | offset_to_ddf_array | Valid | Offset to DDFs from start of header |
| 68-71 | uint32 | offset_to_drf_array | Valid | Offset to DRFs from start of header |
| 72-83 | byte | unknown2 | Unknown | Unidentified |
| **Data Decryption Fields (DDF) and Data Recovery Fields (DRF)** | | | | |
| The fields in the DDF and DRF structures are exactly the same so the structs will be referred commonly as DxF for ease of documentation. The DDF structure appears first followed by the DRF structure. | | | | |
| **DxF Section Header** | | | | |
| **Byte Range (Relative)** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-3 | uint32 | num_dxf_records | Valid | Number of DxF records |
| **DxF Record(s)** | | | | |
| **Byte Range (Relative)** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-3 | uint32 | dxf_record_size | Valid | Size of the DxF record |
| 4-7 | uint32 | pkey_details_offset | Valid | Offset to the struct containing public key information |
| 8-11 | uint32 | efek_size | Valid | Size of the encrypted FEK in bytes. |
| 12-15 | uint32 | efek_offset | Valid | Offset to the Encrypted FEK. |
| 16-19 | uint32 | unknown1 | Unknown | Unidentified |
| 20-variable | struct | PKEY_DETAILS | Valid | |
| variable | byte | efek | Valid | The Encrypted FEK |
| **PKEY_DETAILS** | | | | |
| **Byte Range (Relative)** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-3 | uint32 | pkey_details_size | Valid | Size of the PKEY_DETAILS structure in bytes. |
| 4-7 | uint32 | sid_offset | Valid | Offset to the authorized user's SID |
| 8-11 | uint32 | cred_type | Questionable | 1 = CryptoAPI container, 2 = Unknown, 3 = Certificate thumbprint. |
| 12-variable | byte | cred_value | Valid | Value corresponding to the cred_type. The size of this will be (pkey_details_size-12) bytes. The structure varies based on the type of credential. Full details can be found in the NTFS layout.h file in opensource.apple.com |

Figure D-1: Structure of the $EFS attribute when RSA algorithm is used for asymmetric encryption.

Author Name, email@address

| EFS Stream Header | | | | |
|---|---|---|---|---|
| **Byte Range** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-3 | uint32 | attribute_size | Valid | Size of the entire attribute in bytes |
| 4-7 | uint32 | unknown1 | Unknown | Unidentified |
| 8-11 | uint32 | efsVersion | Questionable | Observed values: 4 |
| 11-15 | uint32 | cryptoAPIVersion | Questionable | Observed values: 0 |
| 16-31 | byte | unknown_hash[16] | Questionable | Possibly MD5 of some struct. |
| 32-35 | uint32 | offset_to_ddf_array | Valid | Offset to DDFs from start of header |
| 36-39 | uint32 | offset_to_drf_array | Valid | Offset to DRFs from start of header |
| 40-43 | uint32 | unknown2 | Unknown | Unidentified |
| 44-45 | uint16 | entity_count_ddf | Questionable | Each DDF/DRF record appears to contain an array of "ENTITY" structures as described below. Number of ENTITY structs in a DDF record. |
| 46-47 | uint16 | entity_count_drf | Questionable | Number of "ENTITY" structures in a DRF record |
| 48-51 | uint32 | alg_id | Questionable | Likely alg_id in wincrypt.h |
| 52-53 | uint16 | struct1_len | Valid | Length-value structure |
| 54-variable | byte | struct1 | Unknown | Unidentified |
| variable | uint16 | struct2_len | Valid | Length-value structure |
| variable | byte | struct2 | Unknown | Unidentified |
| **Data Decryption Fields (DDF) and Data Recovery Fields (DRF)** | | | | |
| The fields in the DDF and DRF structures are exactly the same so the structs will be referred commonly as DxF for ease of documentation. The DDF structure appears first followed by the DRF structure. | | | | |
| **DxF Section Header** | | | | |
| **Byte Range (Relative)** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-3 | uint32 | size_of_all_dxfs | Valid | Size of all DxF records including header in bytes. |
| 4-5 | uint16 | num_dxf_records | Valid | Number of DxF records |
| 6-13 | byte | unknown | Unknown | Unidentified |
| **DxF Record(s)** | | | | |
| **Byte Range (Relative)** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-12 | byte | dxf_record_header | Valid | DxF Record Header |
| 26-variable | struct | DxF_ENTITY | Valid | Each DDF record will have a number of "ENTITY" records as defined in the entity_count_dxf field in the $EFS Section Header. |
| **DxF_ENTITY** | | | | |
| **Byte Range (Relative)** | **Structure** | **Name** | **Validity** | **Notes** |
| 0-1 | uint16 | entity_length | Valid | Length of the ENTITY |
| 2-3 | uint16 | entity_type | Questionable | Type of the ENTITY. 1=Thumbprint, 2=EFEK, 3=User Name, 6=SID, 4= CNG Container Name, 5 = CNG Provider Name |
| 4-5 | uint16 | unknown1 | Unknown | |
| 6-7 | uint16 | unknown2 | Unknown | |
| 7-variable | Unknown | unknown3 | Unknown | Between zero to two uint16 values observed. |
| variable | byte | entity_value | Valid | Value corresponding to entity_type. This is the last field in the ENTITY. |

Figure D-2: Structure of the $EFS attribute when ECC algorithm is used for asymmetric encryption.

Author Name, email@address