Gerben Kleijn (http://gerbenkleijn.com/)

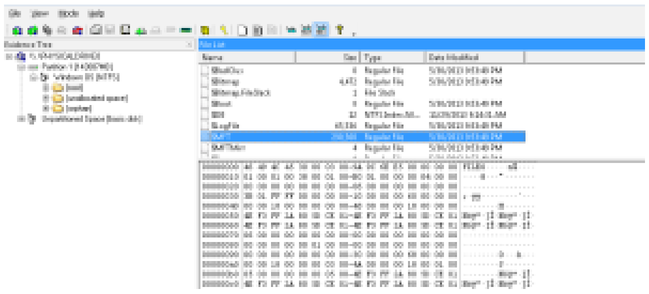
Network Security and Network Engineering

Windows NTFS Master File Table (MFT) Analysis

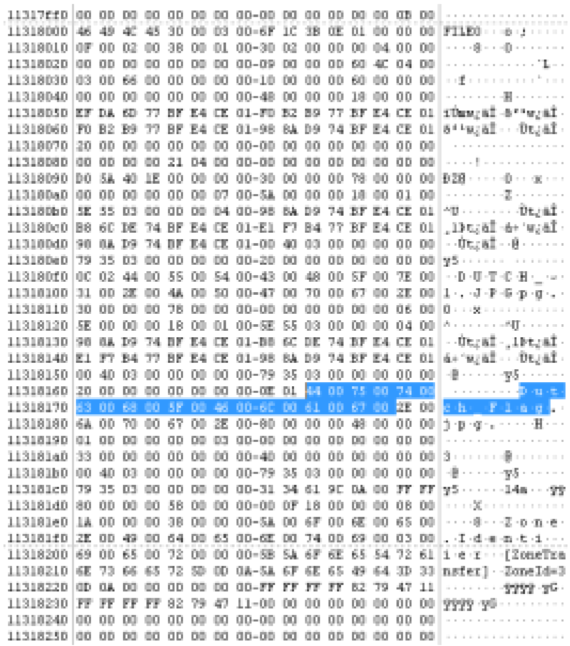
December 2, 2013 (http://gerbenkleijn.com/?p=261) gerbenkleijn (http://gerbenkleijn.com/?author=1) 3 Comments (http://gerbenkleijn.com/?p=261#comments)

In this blog post I will describe how to read file entries in the Master File Table (MFT) for an NTFS volume. I will look at specific sections of hex code for a file and discuss how they relate to the way a file is stored physically on a hard drive. Understanding this post will be easier if you follow along on your own system, but you will need to be able to access your MFT. If you have access to forensic software like EnCase then you probably already know how to access the MFT. If not, then you can download Accessdata's FTK Imager – a free digital forensic software suite that can be downloaded here (http://www.accessdata.com/support/product-downloads). If you know how to use The Sleuth Kit (TSK) to copy the MFT from your hard drive then that is an option too.

The MFT contains information on every file that is saved on the volume and as such, it usually contains tens to hundreds of thousands of entries. To make sure we can find and analyze a known file, download an image file from the Internet and give it a unique name. I downloaded a picture of the Dutch flag to my desktop and named it "Dutch_flag.jpg". Then open up FTK Imager or another forensic software that you choose to use and add your hard drive (the one that contains the image) as an evidence file. The MFT will be located at the root of the drive, for instance at "C:\".


(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic1.png)

Now we need to locate our file in the MFT, which we can do using the search function. Searching for “Dutch_flag” brings me to the following entry in the MFT:

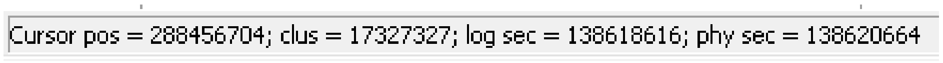


```
11317ff0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 0B 00  ················
11318000  46 49 4C 45 30 00 03 00-6F 1C 3B 0E 01 00 00 00  FILE0···o·;·····
11318010  0F 00 02 00 38 00 01 00-30 02 00 00 00 04 00 00  ····8···0·······
11318020  00 00 00 00 00 00 00 00-09 00 00 00 60 4C 04 00  ············`L··
11318030  03 00 66 00 00 00 00 00-10 00 00 00 60 00 00 00  ··f·········`···
11318040  00 00 00 00 00 00 00 00-48 00 00 00 18 00 00 00  ········H·······
11318050  EF DA 6D 77 BF E4 CE 01-F0 B2 B9 77 BF E4 CE 01  ïÚmw¿äÎ·ð²¹w¿äÎ·
11318060  F0 B2 B9 77 BF E4 CE 01-98 8A D9 74 BF E4 CE 01  ð²¹w¿äÎ···Ùt¿äÎ·
11318070  20 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ················
11318080  00 00 00 00 21 04 00 00-00 00 00 00 00 00 00 00  ····!···········
11318090  D0 5A 40 1E 00 00 00 00-30 00 00 00 78 00 00 00  ÐZ@·····0···x···
113180a0  00 00 00 00 00 00 07 00-5A 00 00 00 18 00 01 00  ········Z·······
113180b0  5E 55 03 00 00 00 04 00-98 8A D9 74 BF E4 CE 01  ^U········Ùt¿äÎ·
113180c0  B8 6C DE 74 BF E4 CE 01-E1 F7 B4 77 BF E4 CE 01  ¸lÞt¿äÎ·á÷´w¿äÎ·
113180d0  98 8A D9 74 BF E4 CE 01-00 40 03 00 00 00 00 00  ··Ùt¿äÎ··@······
113180e0  79 35 03 00 00 00 00 00-20 00 00 00 00 00 00 00  y5··············
113180f0  0C 02 44 00 55 00 54 00-43 00 48 00 5F 00 7E 00  ··D·U·T·C·H·_·~·
11318100  31 00 2E 00 4A 00 50 00-47 00 70 00 67 00 2E 00  1·.·J·P·G·p·g·.·
11318110  30 00 00 00 78 00 00 00-00 00 00 00 00 00 06 00  0···x···········
11318120  5E 00 00 00 18 00 01 00-5E 55 03 00 00 00 04 00  ^·······^U······
11318130  98 8A D9 74 BF E4 CE 01-B8 6C DE 74 BF E4 CE 01  ··Ùt¿äÎ·¸lÞt¿äÎ·
11318140  E1 F7 B4 77 BF E4 CE 01-98 8A D9 74 BF E4 CE 01  á÷´w¿äÎ···Ùt¿äÎ·
11318150  00 40 03 00 00 00 00 00-79 35 03 00 00 00 00 00  ·@······y5······
11318160  20 00 00 00 00 00 00 00-0E 01 44 00 75 00 74 00  ··········D·u·t·
11318170  63 00 68 00 5F 00 46 00-6C 00 61 00 67 00 2E 00  c·h·_·F·l·a·g·.·
11318180  6A 00 70 00 67 00 2E 00-80 00 00 00 48 00 00 00  j·p·g·,·····H···
11318190  01 00 00 00 00 00 03 00-00 00 00 00 00 00 00 00  ················
113181a0  33 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  3·······@·······
113181b0  00 40 03 00 00 00 00 00-79 35 03 00 00 00 00 00  ·@······y5······
113181c0  79 35 03 00 00 00 00 00-31 34 61 9C DA 00 FF FF  y5······14a···ÿÿ
113181d0  80 00 00 00 58 00 00 00-00 0F 18 00 00 00 08 00  ····X···········
113181e0  1A 00 00 00 38 00 00 00-5A 00 6F 00 6E 00 65 00  ····8···Z·o·n·e·
113181f0  2E 00 49 00 64 00 65 00-6E 00 74 00 69 00 03 00  .·I·d·e·n·t·i···
11318200  69 00 65 00 72 00 00 00-5B 5A 6F 6E 65 54 72 61  i·e·r···[ZoneTra
11318210  6E 73 66 65 72 5D 0D 0A-5A 6F 6E 65 49 64 3D 33  nsfer]··ZoneId=3
11318220  0D 0A 00 00 00 00 00 00-FF FF FF FF 82 79 47 11  ········ÿÿÿÿ·yG·
11318230  FF FF FF FF 82 79 47 11-00 00 00 00 00 00 00 00  ÿÿÿÿ·yG·········
11318240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ················
11318250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ················
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic2.png)

Every entry in the MFT is 1,024 bytes in size (1Kb) and starts with the hex value 0×46/49/4C/45/30/, which translates to “FILE0”. Before continuing, make sure that you are looking at the right MFT entry. Even though you only downloaded a single file and gave it a unique name, some operating systems will create a symbolic link for files on the system and this symbolic link will also show up in the MFT. It might have the same file name, but with “.lnk” at the end. If that is the case, keep looking for the original file because otherwise the results from the next steps will not make sense.

FTK Imager will provide you with metadata on sections of the MFT that you are looking at. For instance, if you position your cursor at the very start of the MFT entry for your file you should see a byte offset for your cursor position at the bottom of the screen. The start of my file is at offset 288,456,704 from the start of the MFT.



```
Cursor pos = 288456704; clus = 17327327; log sec = 138618616; phy sec = 138620664
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic3.png)

Since we know that every entry is 1,024 bytes in size, this means that this file is entry number 281,696 in the MFT (288,456,704 / 1,024). This is not very important, but it is always good practice to verify metadata and to know what you are doing and how these numbers relate to each other.

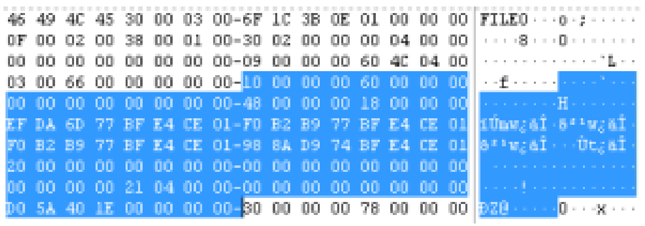
The first piece of hex code that we'll look at are bytes number 22 and 23 from the beginning. You can jump ahead a certain amount of bytes in FTK Imager by using ctrl + g. There are four values that can be found here:

0×00/00	Deleted file
0×01/00	Allocated file
0×03/00	Allocated directory
0×04/00	Deleted directory

I found the value 0×01/00, which means that "Dutch_flag.jpg" is an allocated file on the file system.

Bytes 44-47 of the entry contain the record number. For my file, these bytes had the value 0×60/4C/04/00, in little-Endian encoding. Disregarding the trailing zeros, this makes 0×04/4C/60 in big-Endian encoding. Converting this value to decimal results in the number 281,696, which is exactly the record number that we calculated previously.

The next hex value we'll look at are bytes 56-59 of the MFT entry. These four bytes indicate the Standard Information Attribute marker. An entry in the MFT typically has three of these markers; the Standard Information Attribute, the File Name Attribute, and the Data Attribute. The bytes for the Standard Information Attribute (SIA) should be 0×10/00/00/00.



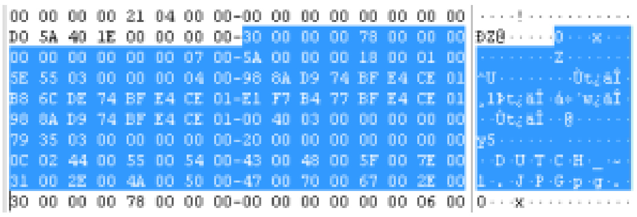
```
46 49 4C 45 30 00 03 00-6F 1C 3B 0E 01 00 00 00  FILE0···o·;·····
0F 00 02 00 38 00 01 00-30 02 00 00 00 04 00 00  ····8···0·······
00 00 00 00 00 00 00 00-09 00 00 00 60 4C 04 00  ············`L··
03 00 66 00 00 00 00 00-10 00 00 00 60 00 00 00  ··f·······`·····
00 00 00 00 00 00 00 00-48 00 00 00 18 00 00 00  ········H·······
EF DA 6D 77 BF E4 CE 01-F0 B2 B9 77 BF E4 CE 01  ïÚmw¿äÎ·ð²¹w¿äÎ·
F0 B2 B9 77 BF E4 CE 01-98 8A D9 74 BF E4 CE 01  ð²¹w¿äÎ···Ùt¿äÎ·
20 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ···············
00 00 00 00 21 04 00 00-00 00 00 00 00 00 00 00  ····!···········
00 5A 40 1E 00 00 00 00-30 00 00 00 78 00 00 00  ·Z@·····0···x···
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic4.png)

The next four bytes, offsets 4-7 from the SIA marker, record the attribute's length. In most cases, only the first of these bytes will be anything but zeros. For my file entry, the value of this byte is 0×60, which translates to 96 in decimal. This means the entire SIA is 96 bytes long. Bytes 16-19 of the SIA indicate the size of the content in the SIA, and bytes 20-23 indicate when the content starts. Again, chances are only bytes 16 and 20 are anything but zeros. In my file entry, these values were 0×48 and 0×18, which converts to 72 and 24 respectively. This means that the SIA content starts at byte 24 from the SIA marker, and the content is 72 bytes in size.

If you jump 24 bytes from the start of the SIA, there should be four 8-byte sequences of data. These sequences are timestamps for the following: (1) Time of file creation, (2) Time when file was last modified, (3) Time when the MFT entry was last modified, and (4) Time when the file was last accessed. There are programs that will decode these values to normal timestamps for you. One such program is Dcode, which can be downloaded for free here (http://www.digital-detective.co.uk/freetools/decode.asp). These values are updated constantly.

These 32 bytes of data are really all the information of interest in the 72 bytes of content contained in the SIA. Immediately after the SIA is the Filename Attribute (FNA) marker, indicated by the value 0x/30/00/00/00. It is not uncommon to find more than one FNA in a file entry. For instance, immediately after the FNA in my file entry there is another 4-byte sequence with a value of 0×30/00/00/00.



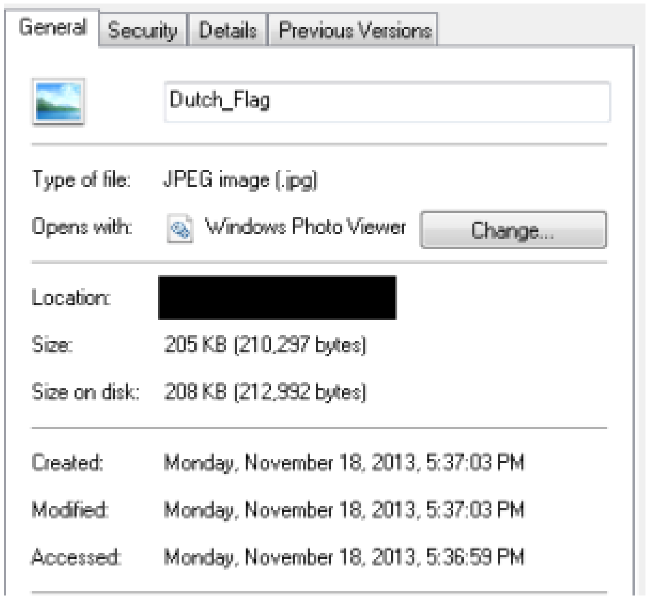
```
00 00 00 00 21 04 00 00-00 00 00 00 00 00 00 00  ....!...........
D0 5A 40 1E 00 00 00 00-30 00 00 00 78 00 00 00  ÐZ@.....0...x...
00 00 00 00 00 00 07 00-5A 00 00 00 18 00 01 00  ........Z.......
5E 55 03 00 00 00 04 00-98 8A D9 74 BF E4 CE 01  ^U........Ùt¿äÎ.
B8 6C DE 74 BF E4 CE 01-E1 F7 B4 77 BF E4 CE 01  ¸lÞt¿äÎ.á÷´w¿äÎ.
98 8A D9 74 BF E4 CE 01-00 40 03 00 00 00 00 00  ..Ùt¿äÎ..@......
79 35 03 00 00 00 00 00-20 00 00 00 00 00 00 00  y5..............
0C 02 44 00 55 00 54 00-43 00 48 00 5F 00 7E 00  ..D.U.T.C.H._.~.
31 00 2E 00 4A 00 50 00-47 00 70 00 67 00 2E 00  1...J.P.G.p.g...
30 00 00 00 78 00 00 00-00 00 00 00 00 00 06 00  0...x...........
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic5.png)

Again, the length of the attribute can be found at bytes 4-7 from the start of the FNA marker. For my file entry, byte 4 contained the value 0×78, or 120 in decimal, meaning the attribute was 120 bytes in size. Bytes 16 and 20 show that the content in the FNA was 0x5A, or 90 bytes in size, and the content started at offset 0×18, or 24. The first 8 bytes of content contain the $MFT record number of the parent (bytes 0-5) and the sequence value of the parent (bytes 6-7).

The next four 8-byte chunks, starting at offset 32 bytes, show the same four 8-byte streams (another 32 bytes) that represent the time and dates relevant to the file. These date-time stamps are redundant to the data in the Standard Information Attribute (SIA) from above. Unlike the timestamps in the SIA, these ones are not updated constantly and are generally not reliable.

The eight bytes after the last 8-byte timestamp entry (offset 40-47) represent the size of the file on disk, in little-Endian encoding. Disregarding the four bytes of training zeros for my file, the value was 0×00/40/03/00, which makes 0×00/03/40/00 in big-Endian encoding. Converted to decimal, this value represents the number 212,992, which is the size on disk in bytes for the file. The size on disk includes file slack – it represents the amount of sectors needed to store the file. Dividing 212,992 by 512 (the size of a sector) shows that 416 sectors are needed to store this file.



Dutch_Flag

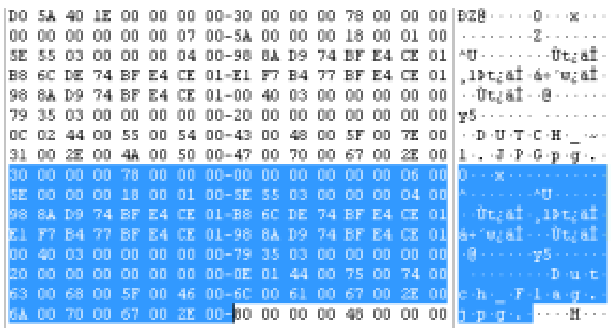
Type of file: JPEG image (.jpg)
Opens with: Windows Photo Viewer
Location: [redacted]
Size: 205 KB (210,297 bytes)
Size on disk: 208 KB (212,992 bytes)
Created: Monday, November 18, 2013, 5:37:03 PM
Modified: Monday, November 18, 2013, 5:37:03 PM
Accessed: Monday, November 18, 2013, 5:36:59 PM

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic6.png)

The actual size of the file without file slack can be found in the next eight bytes. At offset 48-55). Again disregarding the four bytes of trailing zeros, the value for my file was 0×79/35/03/00, or 0×00/03/35/79 in big-Endian. Converting this value to decimal results in 210,297, which is the actual size of the file.

Offset 64 stores the length of the filename and the actual filename is stored beginning at offset 66. The value of byte 64 for my file entry was 0x0C, which is 12. “Dutch_flag.jpg” has 14 characters, but the file name stored in this particular FNA was actually “Dutch_~1.jpg”, which has 12 characters.

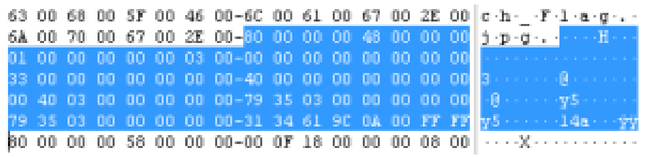
Looking at the second FNA – the 120 byte sequence following the first FNA – we can see that the majority of the byte values are the same but this time the value indicating the length of the filename is 0x0E, which is 14, and the filename is stored as “Dutch_flag.jpg”.



```
D0 5A 40 1E 00 00 00 00-30 00 00 00 78 00 00 00  ÐZ@·····0···x···
00 00 00 00 00 00 07 00-5A 00 00 00 18 00 01 00  ········Z·······
5E 55 03 00 00 00 04 00-98 8A D9 74 BF E4 CE 01  ^U·······Ùt¿äÎ·
B8 6C DE 74 BF E4 CE 01-E1 F7 B4 77 BF E4 CE 01  ¸lÞt¿äÎ·á÷´w¿äÎ·
98 8A D9 74 BF E4 CE 01-00 40 03 00 00 00 00 00  ··Ùt¿äÎ··@······
79 35 03 00 00 00 00 00-20 00 00 00 00 00 00 00  y5··············
0C 02 44 00 55 00 54 00-43 00 48 00 5F 00 7E 00  ··D·U·T·C·H·_·~·
31 00 2E 00 4A 00 50 00-47 00 70 00 67 00 2E 00  1·.·J·P·G·p·g·.·
30 00 00 00 78 00 00 00-00 00 00 00 00 00 06 00  0···x···········
5E 00 00 00 18 00 01 00-5E 55 03 00 00 00 04 00  ^·······^U······
98 8A D9 74 BF E4 CE 01-B8 6C DE 74 BF E4 CE 01  ··Ùt¿äÎ·¸lÞt¿äÎ·
E1 F7 B4 77 BF E4 CE 01-98 8A D9 74 BF E4 CE 01  á÷´w¿äÎ···Ùt¿äÎ·
00 40 03 00 00 00 00 00-79 35 03 00 00 00 00 00  ·@······y5······
20 00 00 00 00 00 00 00-0E 01 44 00 75 00 74 00  ··········D·u·t·
63 00 68 00 5F 00 46 00-6C 00 61 00 67 00 2E 00  c·h·_·F·l·a·g·.·
6A 00 70 00 67 00 2E 00-80 00 00 00 48 00 00 00  j·p·g·.·····H···
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic7.png)

Immediately after the end of the last FNA is the value 0×80/00/00/00, which is the marker for the Data Attribute. Byte four again indicates the length of the attribute. For my file entry this value is 0×48, or 72.

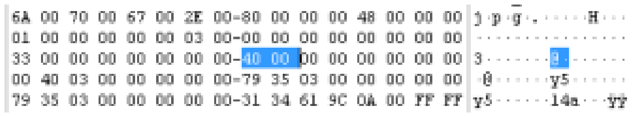


```
63 00 68 00 5F 00 46 00-6C 00 61 00 67 00 2E 00  c·h·_·F·l·a·g·.·
6A 00 70 00 67 00 2E 00-80 00 00 00 48 00 00 00  j·p·g·.·····H···
01 00 00 00 00 00 03 00-00 00 00 00 00 00 00 00  ················
33 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  3·······@·······
00 40 03 00 00 00 00 00-79 35 03 00 00 00 00 00  ·@······y5······
79 35 03 00 00 00 00 00-31 34 61 9C 0A 00 FF FF  y5······14a···ÿÿ
80 00 00 00 58 00 00 00-00 0F 18 00 00 00 08 00  ····X···········
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic8.png)

Eight bytes after the Data Attribute marker there will be a one byte value of either 0×00 or 0×01. If the value is 0×01 that means that the file for this MFT entry is ‘non-resident’ – the data resides somewhere on the hard drive. If the value is 0×00 that means the file is a resident file and the data is actually contained within the MFT entry itself. For certain small files, there is actually enough room in the 1,024 byte MFT entry to store the file data. The value for any image file should be 0×01 because such a file would typically be larger than 1,024 bytes. If the value for your file is 0×00 then you are likely looking at the wrong MFT entry. Also, the results for the next steps will be different if you’re looking at a resident file entry.

Jumping 32 bytes from the start of the Data Attribute marker brings us to a sequence of two bytes that indicate where the ‘data run’ is located. The data run is the sequence of bytes that indicate where the file is actually stored on disk. For my file, the value of these two bytes was 0×40/00 or 0x/00/40 in big-Endian. Converting this to decimal results in 64, meaning the data run is located 64 bytes from the start of the Data Attribute Marker.



```
6A 00 70 00 67 00 2E 00-80 00 00 00 48 00 00 00  j·p·g·.····H···
01 00 00 00 00 00 03 00-00 00 00 00 00 00 00 00  ················
33 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  3·······@·······
00 40 03 00 00 00 00 00-79 35 03 00 00 00 00 00  ·@······y5······
79 35 03 00 00 00 00 00-31 34 61 9C 0A 00 FF FF  y5······14a···ÿÿ
```

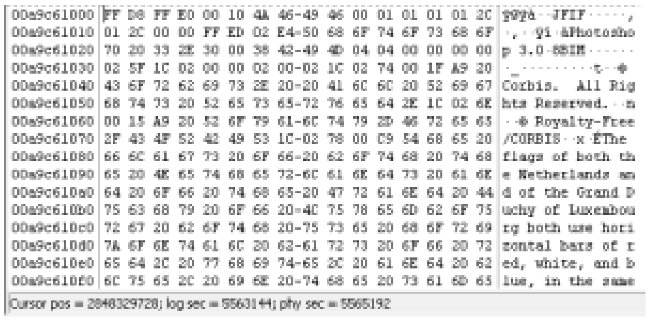
(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic9.png)

Jumping to this location reveals the 8-byte sequence 0×31/34/61/9C/0A/00/FF/FF. The first byte of this sequence is the data run header, which is viewed as two nibbles (3 and 1). Adding these two nibbles together reveals the number of bytes following the header that are used in the data run. For my file that means that four bytes after the header are used for the data run, so we're actually only looking at the value 0×31/34/61/9C/0A. Additionally, the second nibble in the header (1) is used to indicate the number of bytes after the header that are used to indicate how many contiguous clusters are used to store the file data. The first nibble in the header (3) is used to indicate the number of bytes after the header that are used to indicate the starting sector of the file data. If 0×00 follows the run list, then there are no more data runs for this file. If something else than 0×00 follows, then the file is fragmented and spread out over multiple locations on the hard disk. In that case, another data run will immediately follow the first one, indicating the starting point and length of the cluster where the data is continued. This can go on for as many fragments of space are needed to store the file.

For my file, there is only one data run since it is followed by 0×00. After going through this file, I will also provide an example for a file that has multiple data runs. The value 0×34 indicates how many clusters are needed to store the file and the value 0×61/9C/0A indicates the starting sector of the file. Converting these values to big-Endian and then to decimal results in 52 and 695,393. FTK Imager has the ability to 'go to sector/cluster' (ctrl + s), but it only seems to be able to go to a specific sector, not a cluster. Therefore, you want to find out what sector the file is on.

How many sectors go into a cluster depends on what operating system you are using, and what settings you chose when the operating system was installed or when the hard drive was formatted. I believe the Windows 7 default is 8 sectors per cluster. Knowing this, look again at the value that we found previously under the Filename Attribute Marker for the amount of sectors needed to store the file. This value was 416. Divide 416 by 8 and you have 52. Again – it's always good to understand how these numbers fit together.

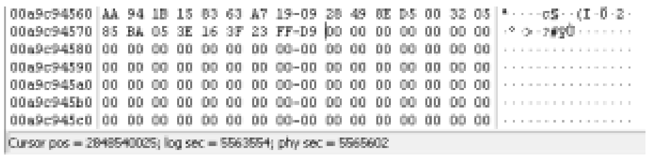
I multiplied the number 695,393 by 8, which results in 5,563,144. In FTK Imager, I navigated out of the MFT and up to the actual partition so that the contents of the entire partition is displayed and then I jumped to sector 5,563,144. This brought me right to the start of the "Dutch_flag.jpg" file.



```
00a9c61000 FF D8 FF E0 00 10 4A 46-49 46 00 01 01 01 01 2C ÿØÿà··JFIF·····,
00a9c61010 01 2C 00 00 FF ED 02 E4-50 68 6F 74 6F 73 68 6F ·,··ÿí·äPhotosho
00a9c61020 70 20 33 2E 30 00 38 42-49 4D 04 04 00 00 00 00 p 3.0·8BIM······
00a9c61030 02 5F 1C 02 00 00 02 00-02 1C 02 74 00 1F A9 20 ·_·········t··©
00a9c61040 43 6F 72 62 69 73 2E 20-20 41 6C 6C 20 52 69 67 Corbis.  All Rig
00a9c61050 68 74 73 20 52 65 73 65-72 76 65 64 2E 1C 02 6E hts Reserved.··n
00a9c61060 00 15 A9 20 52 6F 79 61-6C 74 79 2D 46 72 65 65 ··© Royalty-Free
00a9c61070 2F 43 4F 52 42 49 53 1C-02 78 00 C9 54 68 65 20 /CORBIS··x·ÉThe
00a9c61080 66 6C 61 67 73 20 6F 66-20 62 6F 74 68 20 74 68 flags of both th
00a9c61090 65 20 4E 65 74 68 65 72-6C 61 6E 64 73 20 61 6E e Netherlands an
00a9c610a0 64 20 6F 66 20 74 68 65-20 47 72 61 6E 64 20 44 d of the Grand D
00a9c610b0 75 63 68 79 20 6F 66 20-4C 75 78 65 6D 62 6F 75 uchy of Luxembou
00a9c610c0 72 67 20 62 6F 74 68 20-75 73 65 20 68 6F 72 69 rg both use hori
00a9c610d0 7A 6F 6E 74 61 6C 20 62-61 72 73 20 6F 66 20 72 zontal bars of r
00a9c610e0 65 64 2C 20 77 68 69 74-65 2C 20 61 6E 64 20 62 ed, white, and b
00a9c610f0 6C 75 65 2C 20 69 6E 20-74 68 65 20 73 61 6D 65 lue, in the same
Cursor pos = 2848329728; log sec = 5563144; phy sec = 5565192
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic10.png)

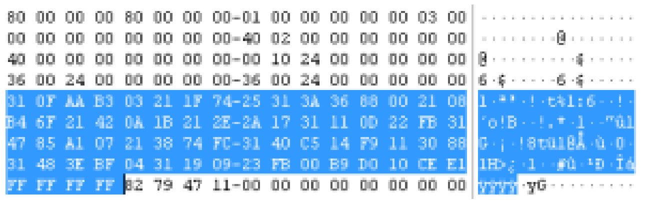
Jumping 210,297 bytes ahead positioned my cursor right behind the value 0xFF/D9 – the footer for the JPEG file signature which indicates the end of a JPEG file. Looking at the metadata at the bottom of the screen revealed that this data was located on sector 5,563,554 – 410 sectors after the starting sector. It makes sense that this number is not 416 because not all of the last cluster was used to store the file. This tells us that the file slack in the last cluster is approximately 6 sectors in size.



```
00a9c94560 AA 94 1B 15 03 63 A7 19-09 28 49 0E D5 00 32 05 ª····c§··(I·Õ·2·
00a9c94570 85 BA 05 3E 16 3F 23 FF-D9 00 00 00 00 00 00 00 ·º·>·?#ÿÙ·······
00a9c94580 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ················
00a9c94590 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ················
00a9c945a0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ················
00a9c945b0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ················
00a9c945c0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ················
Cursor pos = 2848540025; log sec = 5563554; phy sec = 5565602
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic11.png)

So what happens if a file is fragmented and has multiple data runs? As mentioned earlier, the second data run will immediately follow the first one and this will continue for as many data runs as necessary to store the file on disk. Here is an example of a file with 13 data runs.



```
80 00 00 00 80 00 00 00-01 00 00 00 00 00 03 00 ················
00 00 00 00 00 00 00 00-40 02 00 00 00 00 00 00 ········@·······
40 00 00 00 00 00 00 00-00 10 24 00 00 00 00 00 @·········$·····
36 00 24 00 00 00 00 00-36 00 24 00 00 00 00 00 6·$·····6·$·····
31 0F AA B3 03 21 1F 74-25 31 3A 36 08 00 21 08 1·ª³·!·t%1:6··!·
B4 6F 21 42 0A 1B 21 2E-2A 17 31 11 0D 22 FB 31 ´o!B··!.*·1··"û1
47 85 A1 07 21 38 74 FC-31 40 C5 14 F9 11 30 08 G·¡·!8tü1@Å·ù·0·
31 48 3E BF 04 31 19 09-23 FB 00 B9 D0 10 CE E1 1H>¿·1··#û·¹Ð·Îá
FF FF FF FF 82 79 47 11-00 00 00 00 00 00 00 00 ÿÿÿÿ·yG·········
```

(http://gerbenkleijn.com/wp-content/uploads/2013/12/pic121.png)

The first data run indicates that four bytes after the header are used for information on where data is stored and over how many contiguous clusters. Following immediately after the first data run is another data run with the header 0×21, and the one after that has the header 0×31. This continuous until the last data run is encountered which is 0×31/19/09/23/FB. This data run can be identified as the last one because it is immediately followed by the bytes 0×00, indicating the data runs have ended.

So with this file being fragmented across the hard drive, how can the clusters that contain information for this file be found? The process here is a little different from when there is only a single data run. Not for the first data run – that one works exactly the same. However, for the second data run and every one after that, the starting location for the data is actually calculated from the start of the previous data run.

For this example, the first data run (0×31/0F/AA/B3/03) indicates that there are 15 (0x0F) clusters of information stored for this file starting at cluster 242,602 (0×03/B3/AA). Data run two, which is 0×21/1F/74/25, indicates that there are 31 (0x1F) clusters of information starting 9,588 (0×25/74) bytes from the start of the first data run. This would be at cluster 252,190. The third data run indicates that there are 58 clusters of information stored 34,870 clusters from the start of the second data run, which would be cluster 287,060. This continues for all 13 data runs for this file.

However, something strange occurs at data run 7, which is 0×31/11/0D/22/FB. Applying the same process we've been following suggests that there are 17 (0×11) clusters of information stored 16,458,253 clusters from the start of the sixth data run. You can't tell from the picture but the hard drive that was used for this example only contained a total of 1,024,134 clusters, meaning that the designated cluster falls well outside the range of total available clusters.

What happened here is that the 7th data run actually points back to an earlier cluster on the hard drive – one that comes before the 6th data run rather than after it. The key indicator is the last byte of the data run; if the byte falls between the values 0×00 and 0x7F then the amount of clusters to the next data run is positive. If the value of the last byte falls between 0×80 and 0xFF then the amount of clusters to the next data run is negative. For data run 7 the last byte is 0xFB, meaning the amount of clusters to the next data run is negative.

With a data run that points backwards, the process to follow is as follows:

1. Convert the hexadecimal value to Big Endian.
2. Convert the value to binary.
3. Apply a XOR calculation to each bit in the binary value. This means you make each bit the opposite bit, so a 0 becomes a 1 and a 1 becomes a 0.
4. Add 1 to the result.
5. Convert the result to decimal.
6. Subtract the value from the start of the previous data run.

While at first glance this may seem like a crazy amount of work, it's really not so bad. For data run 7, these steps look like this:

1. 0x0D/22/FB = 0xFB/22/0D.
2. 0xFB/22/0D = 1111 1011 0010 0010 0000 1101
3. 1111 1011 0010 0010 0000 1101 becomes 0000 0100 1101 1101 1111 0010
4. 0000 0100 1101 1101 1111 0010 becomes 0000 0100 1101 1101 1111 0011
5. 0000 0100 1101 1101 1111 0011 = 318,963
6. The start of the 6th data run was 328,508, so 328,508 – 318,963 = 9,545.

The start of the 7th data run is at cluster 9,545.

Now that we know how to find out where the clusters with data for this fragmented file are located we can carve (extract, or retrieve) the data. Doing so simply requires you to copy all the data indicated by each data run. So data run 1 indicates that 61,440 bytes (15 clusters; 15 * 4,096 bytes) need to be copied starting at cluster 242,602. Data run 2 indicates that 126,976 bytes need to be copied starting at cluster 252,190 and these bytes need to be appended (added to the end) of the data copied from data run 1. Next, the data from data run 3 is added and this continues until the data for all 13 data runs is added together. If done correctly, the result will be a complete file.

Hopefully this walkthrough made sense. If you have any feedback or if the walkthrough did not match up with what you found on your system, please let me know by leaving me a comment.

3 comments on "Windows NTFS Master File Table (MFT) Analysis"

Gargodong December 24, 2013 11:15 pm

One mistake: Is possible that MFT file may be fragmented, so your formula: Offset / Record size = Record number is not always true.

↳ Reply (/?p=261&replytocom=18#respond)

gerbenkleijn January 6, 2014 3:35 pm

You're absolutely right about that. I'm planning to add a section on analyzing MFT entries of fragmented files – I'll make sure to include your feedback in that!

↳ Reply (/?p=261&replytocom=22#respond)

Yves April 24, 2014 3:06 am

Nice explanation.
About negative data runs you have to check only the first bit of the value (so after big endian conversion)
About fragmented MFT I did some scripting in python in order to rebuild the MFT based on the fact that MFT fragments are always allocated by chunks
When analysing MFT attributes with forensic aims the DATE-TIME values included in FNA's are quite interesting too
Lot of things to discover within resident DATA attributes and unnamed DATA attributes (ADS)

↳ Reply (/?p=261&replytocom=68#respond)

Leave a Reply

Your email address will not be published.

Name*

Email*

Website

Your comment...

You may use these HTML (HyperText Markup Language) tags and attributes:

```
<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite>
<code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>
```

Post Comment

Search …

Search

Recent Posts

Linux IPtables (http://gerbenkleijn.com/?p=289)

Snort Troubleshooting (http://gerbenkleijn.com/?p=284)

Windows NTFS Master File Table (MFT) Analysis (http://gerbenkleijn.com/?p=261)

(http://searchsecurity.techtarget.com/rss/Security-Wire-Daily-News.xml) SearchSecurity (http://rss.techtarget.com/160.xml)

How the Target CEO resignation will affect other execs' security views (http://searchsecurity.techtarget.com/news/2240220103/How-the-Target-CEO-resignation-will-affect-other-execs-security-views)

What should enterprises look for in vulnerability assessment tools? (http://searchsecurity.techtarget.com/feature/What-should-enterprises-look-for-in-vulnerability-assessment-tools)

John Pescatore: BYOIT, IoT among top information security trends (http://searchsecurity.techtarget.com/news/2240220011/John-Pescatore-BYOIT-IoT-among-top-information-security-trends)

(http://seclists.org/rss/bugtraq.rss) Bugtraq (http://seclists.org/#bugtraq)

CVE-2014-0930 - Kernel Memory Leak And Denial Of Service Condition in IBM AIX (http://seclists.org/bugtraq/2014/May/30)

CVE-2014-2882 - Lack of SSL Certificate Validation in Citrix Netscaler (http://seclists.org/bugtraq/2014/May/29)

CVE-2014-2881 - Poor Quality Implementation of Diffie-Hellman Key Exchange in Citrix Netscaler (http://seclists.org/bugtraq/2014/May/28)