

# Detection of Rootkits in the NTFS File System

Mihir Nanavati, Bhavesh Kothari  
MIEL Labs  
MIEL e-Security Pvt. Ltd., Mumbai

## Abstract

*Most malware needs storage space on the disk for its binaries. To prevent signature based detection techniques, the malware often uses rootkits to hide the presence of these files. A number of different techniques are used to hide these files. In this paper, we try and outline all the information necessary to perform cross-view detection of hidden files on an NTFS file system. We also talk of the approach that we have taken with Helios for the detection of hidden files.*

## 1. Introduction

All the files and data present on a disk are stored in the file system. It acts as an index to look up the location of any file on the disk. Most anti-viruses or anti-malware tools rely on the data from the file system while performing a scan. The list of files is obtained from the file system, and then they are scanned for any malicious content. This kind of scan usually involves the use of a signature based detection algorithm.

Conventional anti-viruses are quite effective at detecting and dealing with threats in files, if they are aware of their presence. As a result, rather than just using techniques to make the signature more difficult to detect such as polymorphic code, a large amount of malware is also focusing on hiding the presence of the files altogether. The rootkits that hide the

presence of these files use a variety of techniques to achieve this goal.

One way of hiding files may be to hook the functions that are used to read the file structures from disk such as FindNextFile or Nt/ZwQueryDirectoryFile. Another method does not rely on hooking at all, but use filter drivers to filter the results that are obtained from system calls. Some rootkits also try and hide their presence in the alternate data streams of case of NTFS disks.

File detection techniques have been discussed in the paper about Strider Ghostbuster by Wang, et al.[1] The most commonly used method is termed cross-view detection. In this method, a high level and low level view of the file system are obtained and compared. Discrepancies between these two views are likely the result of rootkit activity and are marked to alert the user. As a result, the scanner does not need to try and worry about exactly how the files have been concealed; it is more interested in the end result that the files have successfully been hidden.

The high level view of the file system is similar to that obtained by tools such as Windows Explorer. It uses the Windows API. The low level view is obtained by parsing the file table structures on the disk.

There are two primary file systems that are used on Windows systems – the older File Allocation Table (FAT) and the newer NT

File System (NTFS). While FAT is a well-documented structure, the exact specification of NTFS has not been provided. In this paper we would like to focus on some of the structures that are necessary to understand to perform low-level scans of an NTFS file system. We would also like to discuss a basic detection technique similar to that we have used in Helios and the test results.

## 2. The NTFS File System

NTFS is the journaling file system introduced in Windows NT and has been used in all subsequent versions. It is increasingly replacing FAT on modern systems.

The entire structure of the NTFS file system is stored in a Master File Table or MFT. This contains a list of all the directories and files on the disk, and the sectors on how to read the data in those files. It also contains the metadata that is supported by NTFS.

One important concept to understand NTFS is that the MFT is made up of fixed size entries known as MFT entries or FILE entries (because of the header tag FILE). Every file on the drive has an MFT entry. The MFT itself also has an MFT entry, i.e. – an entry in the MFT points to itself. This is always the first entry of the MFT. It is, however, not necessary that each file is not limited to a single entry – some files may have multiple MFT entries to store a large amount of information.

The MFT need not be stored in contiguous locations on the disk. Rather, as the number of files increases, the space is allocated in a fragmented manner – just like in the case of any other file on the disk.

### 2.1 MFT Entries

Each MFT entry contains an MFT header, which contains some basic information about the entry and a number of attributes that are associated with that file. It is not necessary that a single MFT entry has only a single attribute of each type. Some entries may be missing some attribute types altogether and may have multiple instances of the same attribute type.

Each attribute itself contains a header which has a common format for all the different attribute types, and some attribute specific data. The attributes can either be resident or non-resident. In resident attributes, the data is stored in the MFT entry itself. This is possible in the case of small attributes. However, in the case of larger attributes (for e.g. consider the data associated with every file), it is not possible to store all the associated data in the attribute itself. These types of attributes are called non-resident attributes. The data of non-resident attributes is stored at different locations on the disk.

### 2.2 NTFS Structures

A detailed and more complete documentation of the different structures are available from the Linux NTFS Project[2]. We will just discuss some of the attributes that are important during detection of rootkits.

**MFT Header:** Every MFT entry starts of with the MFT header. It is distinguished by a signature 'FILE'. It contains an offset to the first attribute present in that entry. This offset is measured from the beginning of the MFT entry. It also contains the Fixup buffer, which is used by the file system to

maintain error checking over sectors (section 2.3.1).

**Attribute Header:** Each attribute starts with an attribute header which signifies the type, name and length of the attribute. It is not necessary that every attribute is named – in fact, majority of the attributes do not have any names. However, names can be used to distinguish between the different attributes of the same types in a single entry. This is particularly useful in case of Alternate Data Streams (section 2.3.2).

As the attributes are stored contiguously in a single entry, the length of one attribute gives us the offset to the next attribute in that entry. The attributes may either be resident or non-resident. In case of a resident attribute, the header contains the offset, from the beginning of the MFT entry, for the value associated with that attribute. In case of non-resident attributes, the header contains the size of the attribute and a run-list of how to read this data from the disk (section 2.3.3 and 2.3.4).

**Attribute List (0x20):** A file that contains a large number of attributes that do not fit in a single entry requires an attribute list. The attribute list is always contained in the first MFT entry linked to a file. It contains an entry for all the attributes, except itself, related to that file, whether or not they lie in that same MFT entry. It also contains a link to the MFT entry that that particular attribute is contained in.

**File Name Attribute (0x30):** This attribute contains the name of the file and some flags. The flags signify whether the file corresponding to the MFT entry is still present on the disk or not. When a file is deleted, it may or may not be removed from the MFT. However, this flag will be

set to signify that it is not in use and can be replaced by the data for another file. The flag also tells is whether the file is a directory or a regular file.

The name of the file is given in different formats in this attribute. The different formats commonly seen are the Win32 format which is the regular file names as seen in Windows Explorer and a DOS name, which follows the old 8.3 naming scheme for large file names. Files with long names contain multiple file name attributes – one for each type format of file name.

**Index Root Attribute (0x90):** This attribute is present in directories, and contains information about the files stored in that directory. It is always resident – in case the directory has more files than can be stored in the entry itself, they are stored in another attribute known as the index allocation attribute. This attribute contains an Index Header, which is followed by a number of Index Entries. Each entry is linked to a single file on the disk.

**Index Header:** It contains the size of the index, and the offset to the first entry from the beginning of the Index Header. As the Index Header is in itself at an offset from the beginning of the attribute, the total offset of the first entry is the total combined offset.

**Index Entry:** It contains the location of the file that it is referring to in the MFT. From this, the MFT entry of that file can be located, and any relevant data for that file can be obtained. It also contains the offset to the next entry and a local file name attribute. Hence, if only the file name is desired, it can be obtained locally rather than retrieving it from the MFT entry in the MFT.

**Index Allocation Attribute (0xA0):** It is for the files that are present in the directory, but did not fit in the Index Root. It is always a non-resident attribute. An Index Block and a number of Index Entries are obtained when this attribute is read from the disk from the run-list.

**Index Block:** The non-resident data of the Index Allocation attribute is a set of one or more fixed size Index Blocks. Each block is distinguished by the starting signature of 'INDX'. It contains a Fixup buffer similar to that of the MFT entry. It is then followed by an Index Header and a number of Index Entries.

**Data Attribute (0x80):** It corresponds to the data of the file. In case of small files, it can be a resident attribute; however, in most cases it is a non-resident one. A file may contain a number of data attributes. The different data attributes may have the same or different names. Each name constitutes one data stream. The primary data stream is the default unnamed attribute. All the attributes with the same name (this includes all the unnamed attributes) form a single stream. The data is read from each of them individually and combined in a buffer to see the resultant data in the file.

## 2.3 File System Operations

### 2.3.1 Fixup Records

Both the MFT entry and the Index block contain a buffer known as the Fixup buffer. It contains a sequence number and a buffer. During writing, the last two bytes of every sector in the record are placed in the buffer and replaced by the sequence number. This is so the sector can later be checked for the integrity of data. During reading, the last two bytes from every sector in the record are compared to the

sequence number. If they match, the data is replaced by the corresponding bytes from the buffer, i.e. – the first sector gets the first two bytes from the buffer, the next the next two and so on.

### 2.3.2 Alternate Data Streams

A file can have a large number of data streams. The default stream is unnamed, but there can also be several named data attributes. These named data attributes are called alternate data streams. Windows Explorer can only see the default data stream of any file. Hence, the alternate data streams can act as a good place to conceal data from the user. These streams are often used to contain some meta-data about the file.

### 2.3.3 Non-Resident Attributes

In case the attributes are non-resident they need to be retrieved by a data run. The attribute has the size of the data that needs to be read by the data run, and the offset to the data necessary for it. The data to be inputted to the data run is called a run list. It contains the offsets to the clusters from which to read data and the number of clusters to read.

### 2.3.4 Data Runs

The data runs require a run list and read the relevant data from the disk. The first byte of the run list can be split into two nibbles. The first nibble gives us the number of bytes required to store the offset, while the second nibble gives us the number of bytes required to store the length. Once the number of bytes representing each value is known, they can be converted to regular numbers. The offset value gives the offset to the cluster from which to start the read operation. This is relative to the current position of the read pointer. The length gives the number of sectors to read from that

position. As an example, let us consider a run list of 21 15 58 11 28 36. The first byte can be split into two nibbles, with two signifying the number of bytes for the offset and the one signifying the number of bytes for the length. Reading the appropriate values, one needs to read 0x15 clusters from the cluster starting at cluster number 0x1158. The next byte is 0x11, which means that both offset and length occupy one byte each. Hence, one needs to further read 0x28 clusters from the cluster number 0x117E (0x1158 + 0x36).

## 2.4 Traversing the File System

The file system starts off by obtaining the entire MFT from the drive. The Fixup operation is performed on the entire MFT. Once this is done, the root directory of the volume is located. This can be identified because the file name attribute for this MFT entry will contain the data ‘.’.

After the MFT entry for the root is found, it has to be parsed for the files and directories it contains. The MFT header gives the offset to the first attribute. The MFT entry is searched for attributes of the type Index Root and Index Allocation. As the MFT entry may contain an attribute list, this too needs to be parsed to look for the specified attributes. In case of Index Root blocks, the data is resident and contains an Index Header and a number of Index Entries. The Index Header will point to the first of several Index Entries. In case of an Index Allocation block, the data is non-resident. After parsing this data by performing the appropriate data runs, the entire attribute is obtained. It contains several Index Blocks.

The Fixup operation is performed on each of the Index Blocks. Each buffer may consist of a number of blocks, the size of

each which is specified in the Index Block structure. Each of these blocks contain an Index Header, which points to a number of Index Entries, just as in the Index Root blocks.

Each Index Entry contains the number of the file record associated with that entry. This can be used to look up the MFT entry of that file. The name and details of the file, including whether it is a directory or a file can be obtained from here.

## 3. Obtaining the Low Level Data of the File System

The low level data can be obtained by reading the MFT using raw disk reads. The first 512 bytes of the volume contain a boot sector. The boot sector contains an offset to the location of the MFT on the drive.

The first entry in the MFT is an entry for the MFT itself. The MFT is like any other file on the disk. Once the Fixup is performed, a data run can be performed to obtain the entire MFT.

The MFT continues to grow as more and more files are added to the volume. This results in the size of the MFT continually increasing and it becomes more and more fragmented. The run-list for the data attribute may become too large to fit in a single MFT entry. In this case, an attribute list is necessary and contains the references to other MFT entries that contain the remaining part of the run-list. The other MFT entries also point into the MFT but may be at locations not read in the first run, i.e. – into the data that is being read in the subsequent data runs. Hence, it is necessary to have a dynamically changing data buffer where the data from each data run is added, and to read each reference to an MFT entry from that buffer.

#### **4. Cross-View Detection in Helios**

Helios performs an online scan of the entire file system on NTFS drives. The “low-level” data is obtained by reading the MFT and parsing the structures into directories and files as described above.

The higher level view is obtained using the FindFirstFile() and FindNextFile() API functions that are used by Windows Explorer to view the file system. The presence of alternate data streams can be checked for either by using the FindFirstStream() and FindNextStream() API functions, or by attempting to open a handle onto a particular stream of a file . Files and alternate data streams that have been hidden will be apparent as they will be absent in the higher level view. Helios will mark them as discrepancies accordingly.

This technique is successful in detecting most of the current rootkits that hide their presence on NTFS file systems. It detects Hacker Defender[3] and Vanquish[4], both of which use hooking to hide their presence and Unreal.A[5] which hides its presence by being in an alternate data stream. Unreal also contains a filter driver on the ntfs.sys which prevents the file system from seeing the presence of that alternate data stream.

#### **5. Limitations**

Any online scan is limited by the fact that the data is being requested and read from a system that may already have been compromised using the same functions that may have already been subverted. While it is not as effective or accurate as an offline scan, the benefits in convenience make it a worthwhile field to investigate.

The file system structures need to be read from the disk from as low a level as possible. Unreal has a filter driver on the ntfs.sys and can be bypassed by directly reading from the disk.sys. However, this becomes an arms war as future rootkits may well filter data from the disk.sys or the classpnp.sys which will result in the detection engine having to go even lower into the kernel to obtain the information. This same rootkit could be more easily detected by finding the driver module that is performing the filtering. Thus, rootkit detection should be developed keeping the entire picture in mind rather than focusing on a particular aspect of it.

#### **6. Conclusion**

In this paper we discussed the storage structure of the file system, and how a rootkit detector could use this understanding of the structure to detect the presence of rootkits and malware. We hope that this information is useful to anyone who is interested either in implementing their own detector, or just in understanding how the NTFS file system works.

We also discussed some of the limitations of cross-view detection with respect to hidden file detection. A large amount of malware also focuses on using ‘Stealth by Design’[6] as opposed to rootkits for achieving their goal of stealth. Any malware detector also needs to evolve with the changing patterns in malware to be successful in coping with the threat posed by it.

## References

- [1] Y.M. Wang, D. Beck, B. Vo, R. Rousev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. Microsoft Research Technical Report MSR-TR-2005-25, February 2005.
- [2] The Linux NTFS Project.  
<http://www.linux-ntfs.org/>
- [3] Holy Father. Hacker Defender.  
<http://www.hxdef.org/>
- [4] EP\_XoFF and MP\_ART. Unreal.A.  
<http://www.rootkit.com/newsread.php?newsid=647>
- [5] XShadow. Vanquish.  
<http://www.rootkit.com/vault/xshadow/vanquish-0.2.1.zip>
- [6] J. Rutkowska. Rootkits vs. Stealth by Design Malware. Black Hat, Amsterdam, 2006.