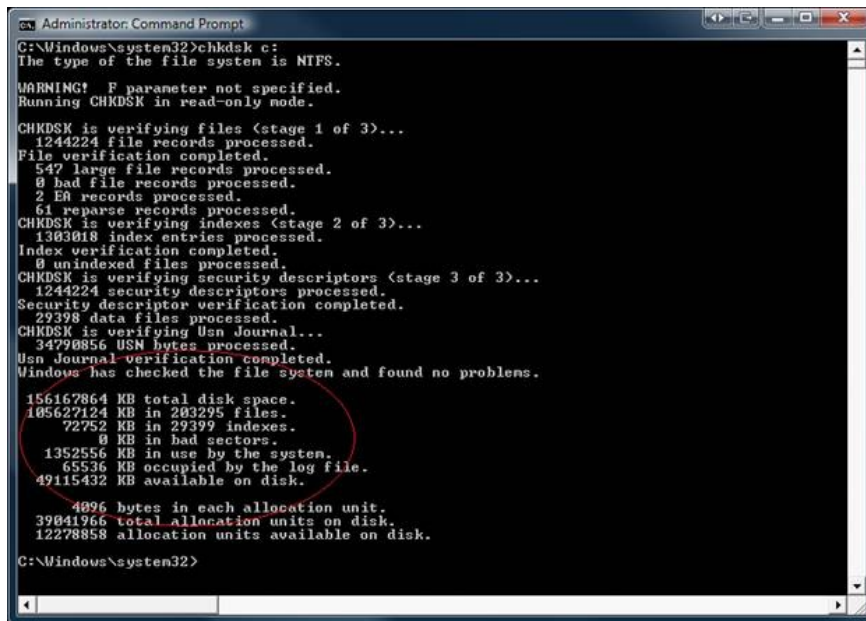


## Ntfs Misreporting Free Space (Part 2)

ntdebug 30 Oct 2008 5:56 PM | 4

Continuing our discussion on the internals of disk usage, we will now shift our focus to internal metadata usage.



```

Administrator: Command Prompt
C:\Windows\system32>chkdsk c:
The type of the file system is NTFS.

WARNING! F parameter not specified.
Running CHKDSK in read-only mode.

CHKDSK is verifying files (stage 1 of 3)...
1244224 file records processed.
File verification completed.
547 large file records processed.
0 bad file records processed.
2 EA records processed.
61 reparse records processed.
CHKDSK is verifying indexes (stage 2 of 3)...
1383818 index entries processed.
Index verification completed.
0 unindexed files processed.
CHKDSK is verifying security descriptors (stage 3 of 3)...
1244224 security descriptors processed.
Security descriptor verification completed.
29398 data files processed.
CHKDSK is verifying Usn Journal...
34798856 USN bytes processed.
Usn Journal verification completed.
Windows has checked the file system and found no problems.

156167864 KB total disk space.
105627124 KB in 283295 files.
72752 KB in 29399 indexes.
0 KB in bad sectors.
1352556 KB in use by the system.
65536 KB occupied by the log file.
49115432 KB available on disk.

4096 bytes in each allocation unit.
39841966 total allocation units on disk.
12278858 allocation units available on disk.

C:\Windows\system32>
  
```

..... KB in .... Indexes.

Consider for a moment a world without indexes... The \$MFT is a database containing records that are accessed via FRS (file record segment) numbers. This FRS number includes an embedded sequence number that is updated anytime a file record is deleted & re-used. A file record must have a new identity once it has been deleted & re-used, so the sequence number is part of this unique identity. Without indexes, you would have to find files by remembering their FRS / Sequence numbers. It would be like remembering your favorite web sites by remembering the IP addresses. In this way, file indexes are like a DNS database so we don't have to find files using FRS numbers. The folder structure has a reserved FRS number for the root. On all NTFS volumes, the root folder is FRS 0x5. Since the root is in a well known location, it can be accessed without doing any index lookups.

Each folder is like a set of DNS records containing information about a domain in the name space. The records contain information about the names and FRS numbers for the files "in" the folder. I put "in" in quotes because the folder itself does not actually contain any files (just records containing basic information about the files). The files are actually records in the MFT that are accessed by FRS number, so the index entries map names to FRS numbers. Folders use two types of metadata streams: \$INDEX\_ROOT:"\$I30" and \$INDEX\_ALLOCATION:"\$I30" to track the names that exist in their namespace. The streams have an attribute type code and a name. For example, \$INDEX\_ALLOCATION is the attribute type, and the attribute name is "\$I30". The "\$I30" name is a short tag indicating that the stream contains file name indexes (as opposed to security indexes, reparse indexes, etc.)

**Why "\$I30"?** Filenames are largely alphanumeric, and the first alphanumeric character in the UNICODE table is 0x30 (48 for those who are hexadecimally challenged). "\$I30" is a shorthand method for saying "Index that's alphanumeric".

When a file is created, the \$FILENAME information is packaged into a name index record which is stored in the parent folder's \$I30 index. With the exception of having one or two \$I30 related streams, there is very little difference between a file and a folder.

Now, let's create a new folder "NewFolder" in the root, and look at the \$I30 index entry created in the root for "NewFolder".

```
D:\>md d:\NewFolder
```

```
D:\>dir d:\
Volume in drive D is d
Volume Serial Number is 4447-4F88
```

```
Directory of d:\
```

```
10/08/2008 02:42 PM <DIR>      NewFolder
0 File(s)           0 bytes
1 Dir(s)           68,620,288 bytes free
```

If you open up an NTFS exploration tool and read the \$INDEX\_ALLOCATION:\$I30 for the root folder, you will find an index entry in the root folder containing the filename "NewFolder". In addition to NFI.EXE, there are some data recovery utilities that can be used to examine NTFS metadata, but I am not able to give any brand names on the blog. NFI.EXE is a useful tool for drilling down into NTFS, and it's **FREE in the OEM Support Tools Phase 3 Service Release 2**. Since NFI is free, it is also not an officially "supported" utility. Despite this, NFI can tell you a lot of information about the allocated ranges of any file. Also, you can give it a logical sector number and it will find the file that owns the sector. For the purpose of this demonstration though, we will be using the standard command line interface "NFI.EXE C:" or "NFI.EXE [drive\path]".

```
C:\Windows\system32>c:\shared\Disktools\nfi.exe d:\
NTFS File Sector Information Utility.
Copyright (C) Microsoft Corporation 1999. All rights reserved.
```

```
Root Directory
  $STANDARD_INFORMATION (resident)
  $FILE_NAME (resident)
  $SECURITY_DESCRIPTOR (resident)
  $INDEX_ROOT $I30 (resident)
  $INDEX_ALLOCATION $I30 (nonresident)
    logical sectors 78920-78927 (0x13448-0x1344f)
  $BITMAP $I30 (resident)
  Attribute Type 0x100 $TXF_DATA (resident)
```

Here is the sector in the root directory \$I30 index allocation that contains our "NewFolder" index entry.

```
LBN 78922
```

```
0x0000 c6 06 3b 47 75 29 c9 01-c6 06 3b 47 75 29 c9 01 |.;Gu) |.;Gu) |.
0x0010 c6 06 3b 47 75 29 c9 01-c6 06 3b 47 75 29 c9 01 |.;Gu) |.
0x0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |.....
0x0030 06 00 00 20 00 00 00 00-07 00 24 00 53 00 65 00 |... ..$.S.e.
0x0040 63 00 75 00 72 00 65 00-0a 00 00 00 00 00 0a 00 |c.u.r.e.....
0x0050 60 00 50 00 00 00 00 00-05 00 00 00 00 00 05 00 |.P.....
0x0060 c6 06 3b 47 75 29 c9 01-c6 06 3b 47 75 29 c9 01 |.;Gu) |.
0x0070 c6 06 3b 47 75 29 c9 01-c6 06 3b 47 75 29 c9 01 |.;Gu) |.
0x0080 00 00 02 00 00 00 00 00-00 00 02 00 00 00 00 00 |.....
0x0090 06 00 00 00 00 00 00 00-07 03 24 00 55 00 70 00 |... ..$.U.p.
0x00a0 43 00 61 00 73 00 65 00-03 00 00 00 00 00 03 00 |C.a.s.e.....
0x00b0 60 00 50 00 00 00 00 00-05 00 00 00 00 00 05 00 |.P.....
0x00c0 c6 06 3b 47 75 29 c9 01-c6 06 3b 47 75 29 c9 01 |.;Gu) |.
0x00d0 c6 06 3b 47 75 29 c9 01-c6 06 3b 47 75 29 c9 01 |.;Gu) |.
0x00e0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |.....
0x00f0 06 00 00 00 00 00 00 00-07 03 24 00 56 00 6f 00 |... ..$.V.o.
0x0100 6c 00 75 00 6d 00 65 00-05 00 00 00 00 00 05 00 |l.u.m.e.....
0x0110 58 00 44 00 00 00 00 00-05 00 00 00 00 00 05 00 |X.D.....
0x0120 c6 06 3b 47 75 29 c9 01-e7 f8 69 2e d8 38 c9 01 |.;Gu) |.
0x0130 e7 f8 69 2e d8 38 c9 01-e7 f8 69 2e d8 38 c9 01 |.i.8f.i.8f.
0x0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |.....
0x0150 06 00 00 10 00 00 00 00-01 03 2e 00 00 00 00 00 |... ..
0x0160 25 00 00 00 00 00 01 00-68 00 54 00 00 00 00 00 |%.....h.T....
0x0170 05 00 00 00 00 00 05 00-36 61 f9 9f 75 29 c9 01 |... ..6a.fu) |.
0x0180 8c 08 e1 8d 61 38 c9 01-8c 08 e1 8d 61 38 c9 01 |i.Bia8f.i.Bia8f.
0x0190 8c 08 e1 8d 61 38 c9 01-00 00 00 00 00 00 00 00 |i.Bia8f.....
0x01a0 00 00 00 00 00 00 00 00-00 00 00 10 00 00 00 00 |... ..
0x01b0 09 00 4e 00 65 00 77 00-46 00 6f 00 6c 00 64 00 |..N.e.w.F.o.l.d.
0x01c0 65 00 72 00 6c 00 75 00-23 00 00 00 00 00 01 00 |e.r.l.u.#.....
0x01d0 88 00 74 00 00 00 00 00-05 00 00 00 00 00 05 00 |e.t.....
0x01e0 f6 7d 8e 47 75 29 c9 01-a6 06 ab 47 75 29 c9 01 |+)AGu) |.
0x01f0 a6 06 ab 47 75 29 c9 01-a6 06 ab 47 75 29 44 00 |*.%Gu) |.

```

Below is the \$I30 index entry in human readable format. Notice that it has everything needed to populate a **WIN32\_FIND\_DATA** structure, and most importantly, the FRS number of our newly created folder. The complete index record contains a duplicate copy of the \$FILE\_NAME attribute from the file record, and this allows [FindFirstFile\(\)](#)/[FindNextFile\(\)](#) to get all pertinent information about our found file without actually opening the file.

```
FileReference      FRS,SEQ <0x25, 0x1> // FRS and Sequence number for "NewFolder"
ParentDirectory    FRS,SEQ <0x5, 0x5> // FRS and Sequence number for the root folder.
CreationTime       : 10/08/2008 LCL 14:40:04.520
LastModificationTime : 10/08/2008 LCL 14:40:04.707
LastChangeTime     : 10/08/2008 LCL 14:40:04.707
LastAccessTime     : 10/08/2008 LCL 14:40:04.707
Allocated Length   : 0
File Size          : 0
File Attribute Flags : 0x10000000 // Attribute flags
File Name          : "NewFolder"
```

Now let's do a pop quiz on indexes to see if everyone is on the same page...

Suppose that you write a fancy new application and you call FindFirstFile() / FindNextFile() in a loop. The cFilename string returned during one of the iterations is "MyFile.txt" (you also have the WIN32\_FIND\_DATA for the same file).

Where did the name "MyFile.txt" come from?

If you call FindFirstFile()/FindNextFile() with a wildcard "\*.\*", is it necessary to open each found file to retrieve the WIN32\_FIND\_DATA?

When you call FindFirstFile(), what is NTFS doing behind the scenes?

What happens when you close the search handle?

#### Answers

If you said "from the file's parent folder's \$I30 index", then you are correct.

If you said "no", then you are correct. The WIN32\_FIND\_DATA is also retrieved from the \$I30 index. There is no need to open the individual files to get this information.

When you call FindFirstFile, NTFS opens the \$I30 index stream(s) for the target folder and scans through the index entries for the first record that matches the specified wild card. A search context is also created to keep track of the current search location in the index stream.

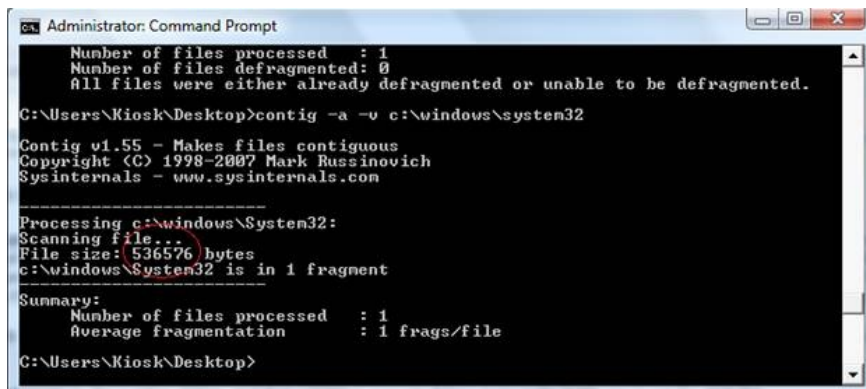
The target folder's index handle is closed and the search context is freed.

If you passed the quiz (or at least understand the answers), you're ready to read on...

In short, high index usage is the result of having a large number of index entries. Common sense would dictate that you probably have the same number of indexes as you have files - Right? Well...the answer is not quite that simple. Suppose that you have 8.3 names turned on and you create a file called "tiny.txt". This file is both 8.3 and LFN compliant, so there will be exactly one index entry created for this file. Now consider what happens when you create a file named "MyFileHasAREallyLongName.txt". This is NOT 8.3 compliant, so NTFS will create an 8.3 name ("MyFile~1.txt"). Now NTFS has to maintain an 8.3 index entry, AND an LFN index entry for a single file. This effectively doubles index usage (plus, long filenames have to be stored in the index and that also makes the LFN filename index larger than normal). If you plan to create a large number of files on a volume, then it is a good practice to either use 8.3 compliant names, or disable 8.3 name creation altogether.

If you have a large folder and want to see how many bytes are in use by indexes, then use contig.exe (from <http://technet.microsoft.com/en-us/sysinternals/bb545046.aspx>) to find out the allocated length of the folder's \$INDEX\_ALLOCATION. Then divide this number by how many files are in the folder. That will give you bytes per index entry.

Below is an example of how to determine how to determine index stream size for a folder.



```

Administrator: Command Prompt
Number of files processed : 1
Number of files defragmented: 0
All files were either already defragmented or unable to be defragmented.

C:\Users\Kiosk\Desktop>contig -a -v c:\windows\system32

Contig v1.55 - Makes files contiguous
Copyright (C) 1998-2007 Mark Russinovich
Sysinternals - www.sysinternals.com

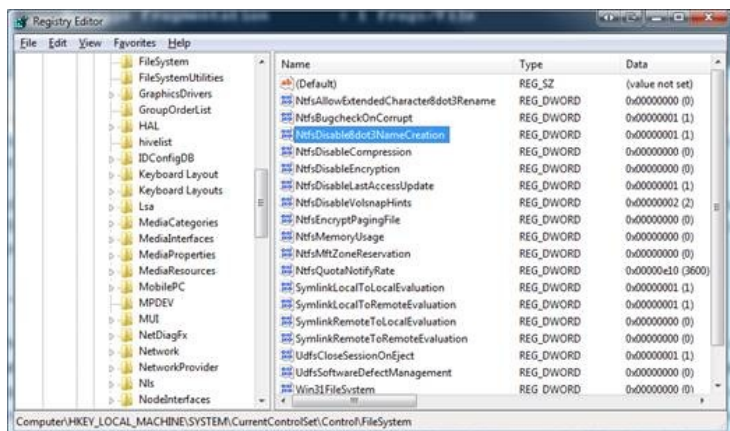
Processing c:\windows\System32:
Scanning file...
File size: 536576 bytes
c:\windows\System32 is in 1 fragment

Summary:
Number of files processed : 1
Average fragmentation : 1 frags/file

C:\Users\Kiosk\Desktop>

```

In my "System32" folder, I had a \$I30 index allocation which was 536,576 bytes long. It contained records for 2,460 files, so this averages out to 218 bytes per index. The presence of 8.3 names can be discovered by running "DIR /X". On my systems, I don't have a need for 8.3, so I turned off 8.3 via the registry (refer to [KB121007](#)).



Whenever possible, try to distribute large numbers of files across several volumes. If you have to put millions of files on a single volume, try to keep your filenames short to save space and improve performance.

..... KB in .... bad sectors.

When a bad sector is detected by CHKDSK /R or if a write occurs because of a bad sector on disk, the cluster that contains the bad sector will be added to the allocated range of \$Badclust. If \$Badclust contains any allocated ranges, then it's time to consider replacing the hard drive.

**IMPORTANT:** If you have a software mirrored volume, and one hard disk has bad sectors, then it is likely that one of the drives in the mirror is going to fail soon. If this happens, keep in mind that when you replace the failing drive, the regenerated mirror set will still have sectors marked in the \$Badclust file even though the mirror is healthy. Since a mirror is a perfect block-by-block copy of the volume, all information for all files is duplicated between the members (including \$Badclust). For this reason, the \$Badclust information is mirrored to the working drive as well as the failing drive.

..... KB in use by the system.

System usage is comprised of \$MFT, \$Logfile, \$Secure, and all other supporting structures in the MFT. If you are looking for system usage, you will need to drill down into the NTFS metadata files.

In most cases, high system usage cannot be "fixed", but it can be kept under control by proper configuration and user education. NFI will give you the information about the size of the various internal metadata files, and you can research the details on how each of the internal system files work, but there simply isn't enough room in a blog post to talk about them all. However, we will discuss the two most common problems that we see: 1. **High \$MFT usage**, and 2. **Bloated Security Stream in the \$Secure File**.

**1. High \$MFT Usage** Every file on the volume is defined by ONE OR MORE file records that are exactly 1KB in size. If the MFT is large, it's because you have a large number of file records in the MFT (free records are also included in the total MFT size). Below are two different ways to view the MFT information. FSUTIL will show you the valid data length, while NFI will give a view of where the fragments of the \$MFT:\$DATA attribute are laid out on disk.

```
Administrator: Command Prompt
C:\shared\Disktools>fsutil fsinfo ntfsinfo c:
NTFS Volume Serial Number : 0x30b86b2bb86aef32
Version : 3.1
Number Sectors : 0x00000000129ddd71
Total Clusters : 0x0000000000253bbae
Free Clusters : 0x0000000000e3069c
Total Reserved : 0x000000000000c60
Bytes Per Sector : 512
Bytes Per Cluster : 4096
Bytes Per FileRecord Segment : 1024
Clusters Per FileRecord Segment : 0
Mft Valid Data Length : 0x000000004bf10000
Mft Start Lcn : 0x0000000000000000
Mft2 Start Lcn : 0x0000000000000010
Mft Zone Start : 0x0000000001b84260
Mft Zone End : 0x0000000001b90b80
RM Identifier: 81DD8C4B-DF20-11DC-8890-806E6F6E963
C:\shared\Disktools>
```

```
Administrator: Command Prompt
C:\shared\Disktools>nfi c:\$MFT
NTFS File Sector Information Utility.
Copyright (C) Microsoft Corporation 1999. All rights reserved.

Master File Table ($Mft)
  $STANDARD_INFORMATION (resident)
  $FILE_NAME (resident)
  $DATA (nonresident)
    logical sectors 6291456-6790239 <0x6000000-0x679c5f>
    logical sectors 179981160-181970823 <0xaba4b68-0xad8a787>
  $BITMAP (nonresident)
    logical sectors 45333416-45333423 <0x2b3bba8-0x2b3bbaf>
    logical sectors 86459968-86459975 <0x5274640-0x5274647>
    logical sectors 95574160-95574167 <0x5b25890-0x5b25897>
    logical sectors 98072222-98072229 <0x5097640-0x5097647>
    logical sectors 101258160-101258167 <0x60913b0-0x60913b7>
    logical sectors 101631416-101631423 <0x60ec5b0-0x60ec5b7>
    logical sectors 98365856-98365863 <0x5dcf1a0-0x5dcf1a7>
    logical sectors 101511504-101511511 <0x60cf150-0x60cf157>
    logical sectors 42063808-42063815 <0x281d7b8-0x281d7bd>
    logical sectors 108052968-108052975 <0x670c1e0-0x670c1e7>
    logical sectors 41314336-41314343 <0x2766820-0x2766827>
    logical sectors 4034784-4034791 <0x3d90e0-0x3d90e7>
    logical sectors 7942192-7942199 <0x793030-0x793037>
    logical sectors 98365824-98365831 <0x5dcf180-0x5dcf187>
    logical sectors 108726976-108726983 <0x67c1c30-0x67c1c37>
    logical sectors 93666368-93666375 <0x5953c40-0x5953c47>
    logical sectors 99010480-99010487 <0x5e6c7b0-0x5e6c7b7>
    logical sectors 116242000-116242007 <0x6edb650-0x6edb657>
    logical sectors 119205928-119205935 <0x71af028-0x71af02f>
    logical sectors 125592304-125592311 <0x77c62f0-0x77c62f7>
    logical sectors 128631096-128631103 <0x7aac130-0x7aac137>
    logical sectors 131908984-131908991 <0x7dec578-0x7dec57f>
    logical sectors 133267864-133267871 <0x7f18198-0x7f1819f>
    logical sectors 133512568-133512575 <0x7f53d78-0x7f53d7f>
    logical sectors 135797648-135797655 <0x8181b90-0x8181b97>
    logical sectors 136190432-136190439 <0x81e19e0-0x81e19e7>
    logical sectors 137952088-137952095 <0x8990be0-0x8990be7>
    logical sectors 141520176-141520183 <0x86f6d30-0x86f6d37>
    logical sectors 149870384-149870391 <0x8eed730-0x8eed737>
    logical sectors 161017672-161017679 <0x998ef48-0x998ef4f>
    logical sectors 182450768-182450775 <0x61b4650-0x61b4657>
    logical sectors 191947464-191947471 <0xb70e2c0-0xb70e2cf>
    logical sectors 242057616-242057623 <0xe6d370-0xe6d377>
    logical sectors 245232152-245232159 <0xe9df218-0xe9df21f>
    logical sectors 179649760-179649767 <0xab53ce0-0xab53cef>
    logical sectors 190010224-190010231 <0xb535370-0xb535377>
    logical sectors 126059120-126059127 <0x7838270-0x7838277>
C:\shared\Disktools>
```

Unfortunately, if your \$MFT is too big and you want it to be smaller, you will have to reformat the drive. Just keep in mind that once you restore your files, you will have 1KB of MFT allocated for each file on the drive (lots of extra file records are needed to restore your 20GB compressed files), but I will assume that everyone read part 1 and they are not going to do that. C);3)

## 2. Bloated Security Stream in the \$Secure File

Following good development practices will save you lots of headaches with your \$Secure file. If you write applications that ACL & re-ACL files over and over and over, your \$Secure file probably looks like mine (love those logon scripts from the IT department)...



```

Administrator: Command Prompt
File 9
Security ($Secure)
$STANDARD_INFORMATION (resident)
$ATTRIBUTE_LIST (nonresident)
    logical sectors 3537832-3537839 {0x35fba8-0x35fbaf}
$FILE_NAME (resident)
$INDEX_ROOT $SDH (resident)
$INDEX_ROOT $SII (resident)
$INDEX_ALLOCATION $SDH (nonresident)
    logical sectors 5221488-5221493 {0x5c6f00-0x5c7277}
    logical sectors 52276504-52276511 {0x31dad18-0x31dad1f}
    logical sectors 52268792-52268799 {0x31d8ef8-0x31d8eff}
    logical sectors 52276192-52276199 {0x31dabe0-0x31dabe7}
    logical sectors 52281568-52281575 {0x31dc0e0-0x31dc0e7}
    logical sectors 54981752-54981759 {0x345b1b8-0x345b1bf}
    logical sectors 54987288-54987295 {0x3470a18-0x3470a1f}
    logical sectors 54987304-54987311 {0x3470a28-0x3470a2f}
    logical sectors 550858056-550858063 {0x3481e88-0x3481e8f}
    logical sectors 55066760-55066767 {0x3484088-0x348408f}
    logical sectors 55068680-55068687 {0x3484808-0x348480f}
    logical sectors 55073656-55073663 {0x3485708-0x348570f}
    logical sectors 52187976-52187983 {0x31c5348-0x31c534f}
    logical sectors 52181936-52181943 {0x31c3bb0-0x31c3bb7}
    logical sectors 15996000-15996007 {0xf41460-0xf41467}
    logical sectors 15403952-15403959 {0x6b0bb0-0x6b0bb7}
    logical sectors 15365240-15365247 {0x62470-0x62477}
    logical sectors 47480392-47480399 {0x2d47e8-0x2d47ef}
    logical sectors 47459088-47459095 {0x2d42b0-0x2d42bf}
    logical sectors 113682160-113682167 {0x6c6a6f0-0x6c6a6f7}
    logical sectors 113691144-113691151 {0x6c6ca08-0x6c6ca0f}
    logical sectors 113691464-113691471 {0x6c6cb48-0x6c6cb4f}
    logical sectors 113692000-113692007 {0x6c6cd08-0x6c6cd0f}
    logical sectors 113691656-113691663 {0x6c6cc08-0x6c6cc0f}
    logical sectors 113691456-113691463 {0x6c6cb40-0x6c6cb47}
    logical sectors 113691240-113691247 {0x6c6ca68-0x6c6ca6f}
    logical sectors 113691136-113691143 {0x6c6ca00-0x6c6ca0f}
    logical sectors 113229872-113229879 {0x6bf030-0x6bf037}
    logical sectors 113252560-113252567 {0x6c018d0-0x6c018df}
    logical sectors 113259336-113259343 {0x6c03348-0x6c0334f}
    logical sectors 113259368-113259375 {0x6c03368-0x6c0336f}
    logical sectors 113242768-113242775 {0x6bf290-0x6bf297}
    logical sectors 212221608-212221615 {0x6bae28-0x6bae2f}
    logical sectors 212221608-212221799 {0xcaddc8-0xcade07}
$INDEX_ALLOCATION $SII (nonresident)
    logical sectors 80-87 {0x0-0x57}
    logical sectors 285988-285103 {0x459a0-0x459af}
    logical sectors 297544-297847 {0x484a8-0x48477}
    logical sectors 30597236-30597239 {0x12f000-0x12f007}
    logical sectors 248198800-248199055 {0xcxc369-0xcxc378f}
$BITMAP $SDH (resident)
$BITMAP $SII (resident)
File 10
Upcase Table ($UpCase)
$STANDARD_INFORMATION (resident)
$FILE_NAME (resident)
$DATA (nonresident)
    logical sectors 140776-141031 {0x225e8-0x226e7}

```

To those of you who are savvy with file system internals, you probably noticed something was missing in the picture above. The \$Secure file shows \$SII/\$SDH INDEX ROOT(s)/INDEX ALLOCATION(s), but where is the actual security stream \$DATA:\$SDS?

There simply was no more room in the base file record for the \$DATA:\$SDS attribute, so it was moved to a child record. To find the child record, we can read the \$ATTRIBUTE\_LIST (via sector editor) and find the pointer to the file record(s) that hold the \$SDS stream metadata. To keep the legal department happy, I can't give you the data types, but I can tell you that my \$DATA:\$SDS stream (shown below) is split between two child records because the \$SDS stream is heavily fragmented. The first child record is FRS 0x1888, and the other is FRS 0x12d7e. If you were to read those two file records, they would each contain the mapping information for approximately 400 fragments of my security stream.

LBN 3537832

```

0x0000 10 00 00 00 20 00 00 1a-00 00 00 00 00 00 00 00 ▶...→.....
0x0010 09 00 00 00 00 00 09 00-00 00 d4 00 09 00 00 00 .....↳.....
0x0020 30 00 00 00 20 00 00 1a-00 00 00 00 00 00 00 00 0...→.....
0x0030 09 00 00 00 00 00 09 00-07 00 24 04 53 65 53 63 .....$.SeSc
0x0040 80 00 00 00 28 00 04 1a-00 00 00 00 00 00 00 00 Ç...(!→.....
0x0050 88 18 00 00 00 00 37 00-00 00 24 00 53 00 44 00 ê?...7...$.S.D.
0x0060 53 00 02 00 01 01 00 00-80 00 00 00 28 00 04 1a S.....Ç...(!→
0x0070 f9 01 00 00 00 00 00 00-7e 2d 01 00 00 00 0f 00 .....~.....
0x0080 00 00 24 00 53 00 44 00-53 00 00 00 00 00 00 00 ..$.S.D.S.....
0x0090 90 00 00 00 28 00 04 1a-00 00 00 00 00 00 00 00 É...(!→.....
0x00a0 09 00 00 00 00 00 09 00-60 17 24 00 53 00 44 00 .....↑$.S.D.
0x00b0 48 00 00 00 00 00 00 00-90 00 00 00 28 00 04 1a H.....É...(!→

```

My security stream may look scary because it has 400 fragments, but it is only about 3.3MB plus the size of the \$SII & SDH streams. If it were to grow past the 1GB range, I would start looking for the cause of the growth.

In theory, you can bloat your \$SDS stream by creating lots of unique security descriptors, but this is usually not the cause of bloating. Instead, most mischief is caused by application developers who call `SetFileSecurity()` without properly preparing their security descriptor buffer.

Most applications:

1. Allocate some heap memory.
2. Init the SD via `InitializeSecurityDescriptor()`.
3. Set up the ACE's.
4. Assign security to the target object.

The problem is that heap memory is like recycled paper. When you call `InitializeSecurityDescriptor()` the first few bytes of your buffer will say “I’m a security descriptor”, but the ending bytes will have some text from an e-mail you decided not to send to your boss. As the SD is filled in with ACE’s, the letter to the boss is overwritten with the ACE’s. At that point, your buffer looks like a valid SD to the system, but there’s still some slack space at the end that says *“Porsche destroyed in the fire. Yours truly, Larry”*. When you send this buffer to `SetFileSecurity()`, NTFS takes this buffer and computes a hash value to determine

whether this SD is unique (the salutation to your boss is also included in the hash). If the hash is identical to a hash value in the \$SDH stream, then we do a comparison between the new & existing SD's. If they match a byte-per-byte comparison, then the existing SD is used. If not, your new SD is added to the stream (along with the bad news about the boss' car). To prevent this, always zero your entire security descriptor buffer prior to calling InitializeSecurityDescriptor(). You will prevent \$SDS bloating and your boss will never know about the car.

I hope you all find this information useful in your sleuthing efforts.

Best regards,

Dennis Middleton "*The NTFS Doctor*"

## Comments



**ИТ.ИБ и т.п.**

31 Oct 2008 3:40 PM

Так как начинается горячая пора, связанная с подготовкой к Платформе, то я вынужден буду на этот месяц...



**Wampiryczny blog**

20 Nov 2008 5:34 PM

Po dłuższym przestoju serwer się ocknął, zobaczymy na jak długo. W międzyczasie na wiki opisałem krótki eksperyment odnośnie nagłówka Cache-Control w Internet Explorer oraz Firefox. Jego rezultaty w skrócie - lepiej stosować no-store niż no



**Nik**

2 Jan 2009 12:28 PM

Also there might be and overhead for the object identifiers. (OBJID file and per-record overhead)



**Gene**

12 Apr 2011 4:35 PM

Just a side note, found an interesting hotfix from Microsoft for Chkdsk which can compact the security descriptor database. KB article # 919241.

Had a system in use of 1.6GB on a 10GB partition. Found your article and then with further digging found the hotfix from MS, thought I would post in response.