Global Information Assurance Certification Paper

Alternate Data Streams: Out of the Shadows and into the Light

by

Ryan L. Means

for GCWN v3.1 (Option 3)

Abstract

Alternate Data Streams: Out of the Shadows and into the Light examines alternate data streams in NTFS. It provides a thorough technical background in alternate streams before proceeding to compare them to regular files and directories. There is then a study of several techniques by which alternate data streams can be exploited by malicious users. The paper then examines software from Microsoft and third-party vendors, evaluating each application's effectiveness in finding and manipulating alternate data streams. Finally, the paper presents a set of Windows shell extensions designed to make alternate stream information an integral part of the operating system and eliminate a loophole that malicious users can use to hide alternate data streams from current scanners.

What is an alternate data stream?

The simplest definition of an alternate data stream in NTFS is:

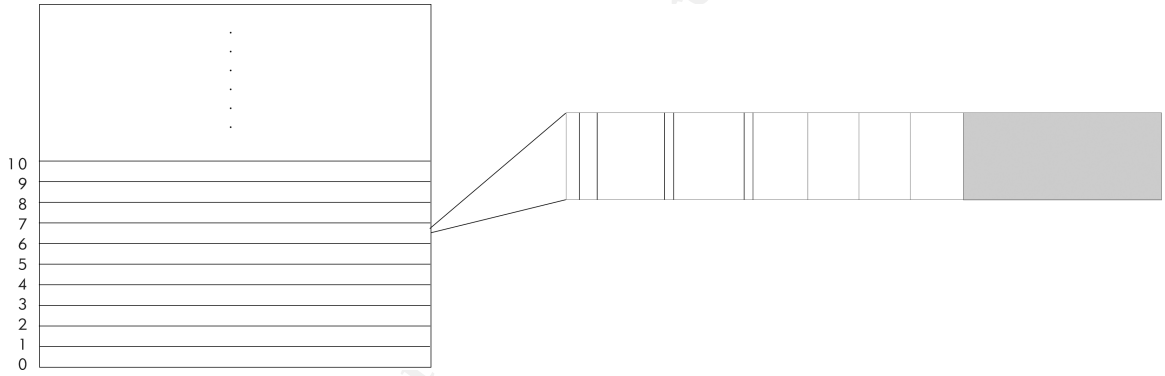
The stream in any data attribute on a file or directory other than the default, unnamed stream.

Understanding this definition requires knowing the structure of a special metadata file called the Master File Table (MFT). The system creates twelve metadata files when an NFTS 5.0 partition is formatted that contain information about the volume itself and the data stored on it. The first metadata file, the MFT, stores all of the records and attributes that Windows needs to access any file or directory on the volume.

Structure of the Master File Table (MFT)

The Master File Table stores a list of records containing attributes as illustrated in Figure 1.

Figure 1: The MFT as a list of records



Attributes consist of distinct groupings of data in the record that conform to a certain structure. Each attribute type stores a different piece of information about the file or directory, ranging in scope from a single bit to indicate a read-only document to the full content of a video file just one kilobyte shy of sixteen exabytes, the theoretical maximum file size on NTFS[1]. The size of each of the records in the MFT equals the cluster size of the volume, with a minimum of 1,024 bytes and a maximum of 4,096 bytes[2]. The structure of the file record follows:

Structure of a Record in the MFT

First, the record starts out with a header that contains, among other data-integrity information, a pointer to the first attribute in the record. The individual attributes immediately follow the record header. NTFS has over a dozen system

[1] "Size Limitations in NTFS and FAT File Systems." Microsoft TechNet.
[2] Kozierok, Charles M. "Master File Table (MFT)."

defined attributes and the ability to have any number of user-defined attributes, but understanding alternate data streams requires knowledge of only the following three:

- **Standard Information Attribute (SI):** Contains file creation time, modification time, last access time, and standard DOS file permissions (System, Hidden, Archive, Read-Only, etc.) Each record can only have one SI attribute.
- **File Name Attribute (FN):** Contains a name for the file or directory. This name can be the regular UNICODE name, the MSDOS 8.3 short filename, or any other identifying data. Multiple file name attributes can exist in the record to hold each of these possible names.
- **Data Attribute (DATA):** Contains the data for the file. There can be multiple data attributes in each record.

Each attribute in the MFT record will be divided into two parts, the header and the content:

Attribute Headers

There can be four different types of headers for an attribute, depending on whether the attribute has a name or not and whether the content part is stored immediately following the header in the MFT (resident) or in an alternate location on the volume (non-resident). All four header types store information including the type of attribute (SI, FN, DATA, etc.), the attribute length, the residency of the content portion, and in the case of DATA attributes, the compression status of the content. Additionally, for resident content, the header contains data about the length of the content. However, in the case of non-resident content, the header stores information identifying the portion of the data referenced in the content as well as its size.

Attribute Content

The structure of the content portion depends on the size of the data. When appropriately sized, the content exists immediately following the header in the MFT record itself. If the amount of data to be stored exceeds the remaining size of the record, the content portion contains a *runlist* of pointers to the actual location of the data elsewhere on the disk. Every piece of data on the volume is stored in a cluster, the smallest unit of allocation on the disk. The runlist stores a list of elements that contain both the Logical Cluster Number (LCN), or the starting cluster for a set of data, and the number of clusters in that particular *run*. It usually takes many runs to store all of the data because of disk fragmentation, so the number of elements in the runlist may be large. The collection of content in the data attribute, whether resident in the MFT or non-resident in clusters identified by the runlist, is called the stream.

To illustrate the structure of the file record that has been outlined so far, see Figure 2.

Figure 2: A file record in the MFT

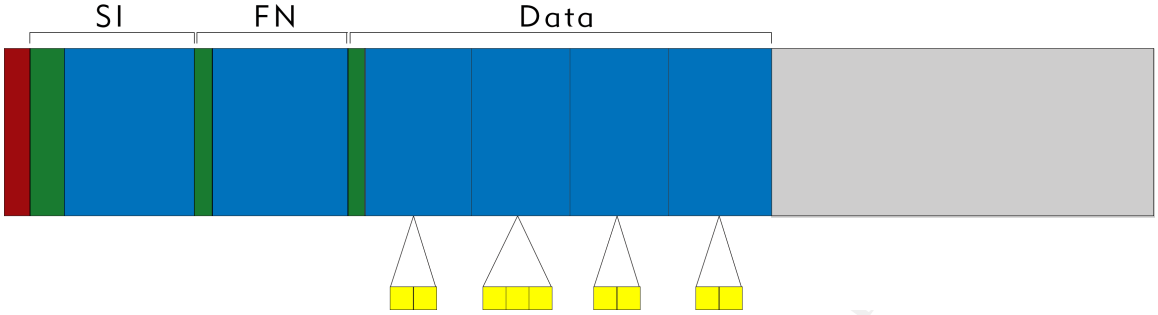
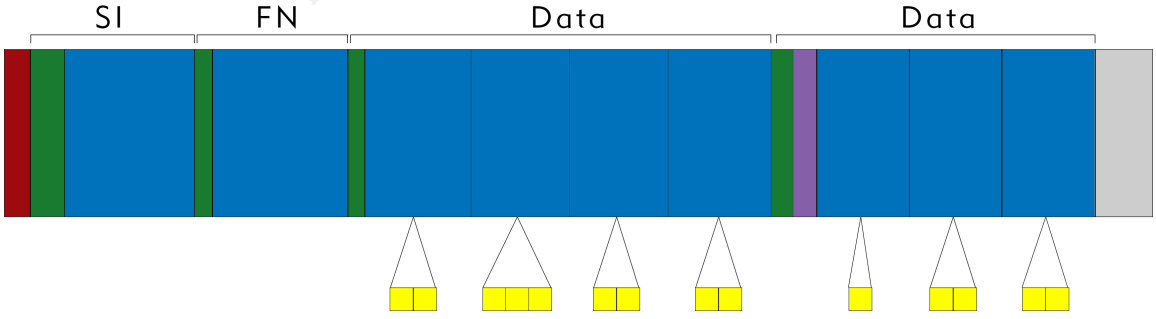


Figure 2 shows a record in the MFT with the record header in red, and each of the three attributes (Standard Information, File Name, and Data) with a green header and blue content. This particular record contains a non-resident stream comprised of clusters stored at various locations on the volume. Each element of the blue runlist in the Data attribute identifies the set of yellow clusters containing the stream data. If the stream were small enough to be resident, the runlist would be replaced by data, immediately following the green header. On the other hand, if the runlist gets too large to fit in the record, NTFS has a mechanism to store it in one or more external attributes in a separate MFT record.

Structure of a Record with an Alternate Data Stream

In a file with an alternate data stream, the record looks exactly the same as the one depicted in Figure 2 except instead of having only one data attribute it has two or more. NTFS allows only one unnamed data attribute per record, so any additional data attributes must be named. These additional named data attributes contain the alternate data streams. The structure of a file record with an alternate data stream is illustrated in Figure 3.

Figure 3: A file record with an alternate data stream



This record looks very similar to the record in Figure 2, with the addition of a second data attribute. This second data attribute has its name stored in the

purple area, which is at the end of the attribute header. The second data stream references its own clusters completely separate from the first.

Directories and Alternate Data Streams

Directories are also represented within records in the MFT. Directories use another type of attribute, the Index Root attribute, which stores an index to the files in the particular directory very much like the way a data attribute stores a stream as outlined above. However, just because a record contains this Index Root attribute does not mean that it cannot also contain one or more data streams. Because a directory record does not usually have a data attribute, any data attribute present in a directory record is automatically an alternate data stream. Figure 4 shows a MFT record for a small directory with a resident index and an alternate data stream.

Figure 4: A directory record with an alternate data stream

This directory record's data attribute, like that of the file records in Figure 2 or 3, points to clusters elsewhere on the volume. Additionally, all data attributes on a directory record must be named, so the purple area at the end of the data attribute's header indicates the presence of an attribute name.

How are alternate data streams used legitimately?

NTFS implemented alternate data streams with the advent of Windows NT 3.1 in May of 1993 and successive versions of Windows have utilized them more and more. They have a host of legitimate uses on both the server and workstation including, but not limited to, the following:

Services for Macintosh

Alternate data streams were initially created for Windows NT 3.1 so that NT, using NTFS, could act as a file server for Macintosh clients. The Macintosh's Hierarchical File System (HFS) had long stored files in two "forks", the data fork and the resource fork. On the Mac, the data fork stores the application data while the resource fork contains supporting metadata like icons or locale information. Using Services for Macintosh, a Windows file server can keep the data fork in the primary, unnamed stream and the resource fork in an alternate data stream. This maintains the transparency of file manipulation on a Macintosh even if it's using files on a Windows share.

Storing "Summary" Data

With the introduction of Windows 2000, Microsoft began to offer a "Summary" tab that shows up in the properties of a selected file. Some of the summary information displayed in the tab includes title, subject, and author. This summary metadata is stored in an alternate data stream of the selected file. In the case of image files, a thumbnail can also be cached in the summary metadata to speed up the display images when the Explorer is in thumbnail view.

Volume Change Tracking

Also introduced in Windows 2000, Microsoft decided to implement volume change tracking NTFS volumes. Volume change tracking allows an application to monitor changes to a volume and take action if necessary. This is very useful for virus-scanners, backup applications, and Microsoft's File Replication Service which is used by a Distributed File System (DFS). Rather than scanning the entire file system to look for modifications, these applications can be notified by NTFS on updates to the change journal. The change journal is stored in an alternate data stream in one of the twelve special metadata files that are created when a volume is formatted[3]. This metadata file stores information about the name of the modified file along with information about what specific change was made.

[3] Russinovich, Mark. "Inside Win2K NTFS, Part 2."

How are alternate data streams handled differently from regular files or directories?

Alternate data streams, though only slightly different from primary data streams, are handled very differently both by Microsoft and third-party applications on Windows. The biggest differences between primary and alternate data streams have to do with whether or not an application knows alternate streams exist, and if it does, how they are accessed. The data that exists in an alternate stream cannot be deleted in the same manner as data in a primary stream, rendering the standard Microsoft tools useless if the content needs to be removed. Finally, each data stream possesses its own lock attributes, but Windows only pays attention to the lock on the unnamed stream. This characteristic creates a very interesting vulnerability in which alternate data streams can be created and edited while being protected from discovery or removal by alternate data stream scanning applications.

Most applications do not appear to recognize alternate data streams

There are at least three groups of applications that *should* recognize alternate data streams and do not. System default tools like Windows Explorer and the command shell, virus scanners, and file integrity programs all fail to behave appropriately when an alternate data stream is present.

Windows Tools

The tools that Microsoft provides with their Windows operating system would be the most obvious place that one would expect to find information about a file's alternate data streams. Unfortunately, this functionality has yet to materialize. Even the latest versions of Windows still do not, by default, provide the user any clue as to the existence or size of an alternate data stream. A detailed file or directory listing in Windows Explorer looks exactly the same regardless of whether a file has no alternate data streams, one five byte stream, or eighty 600 megabyte streams. In addition to the lack of visual cue as to the presence of another stream, there is no way to view the contents of that stream in the GUI. The latter portion of this document will include a set of shell extensions that fixes this, the most severe problem with the implementation of alternate streams in Windows.

It would be reasonable to assume that streams would be left out of Explorer to simplify the interface for the majority of users but would be available to power users willing to find them through the standard command-line applications. However, this is also not the case. Regardless of the options used, the supplied application for listing the contents of a directory, `dir`, will not reveal the presence of an alternate data stream. Furthermore, `dir` ignores any named streams in files or directories when calculating the total size of the file.

Some command line tools provided with Windows and with the Windows Resource Kit allow you to copy data into and out of a named stream on a file or

directory. These tools, however, are of limited effectiveness due to their inability to enumerate the streams in a file. Microsoft's tools enable you to store data in an alternate stream and get it back out again, but only if you can remember the name of the stream you put it in.

In a strange deviation from the norm, Microsoft did give their Notepad application the ability to read and write named streams. But, like the command-line tools above, Notepad does not have the ability to enumerate the streams that already exist in a file or directory.

Virus Scanners

Alternate data streams have been around since the early 1990's, but applications from major virus-protection vendors still fail to recognize their presence under certain circumstances as indicated by a report from Dartmouth's Institute for Security Technology Studies[4]. Virus scanners do not enumerate the alternate streams on a file as part of a normal on-demand scan. Virus-protection software vendors claim that any viruses will be detected by their real-time monitoring software, which may be true; nevertheless, this solution is far from complete. Once a virus has been detected, a virus scanner that does not understand named streams will not be able to disinfect the file and the user will be forced to revert to third-party tools to remove the stream or to delete the entire file outright.

File Integrity Software

File integrity software is designed to create a unique value for a particular file that is based on the data in that file. This value can then be stored or distributed as a way of verifying that the contents of the file have not been changed. Applications such as md5sum and cksum use Message Digest 5 (MD5) or Cyclic Redundancy Checking (CRC) algorithms, respectively, to calculate the unique integrity value for a given file. Windows ports of the GNU versions of these applications obtained from http://unxutils.sourceforge.net/ yielded the following results:

The file `test.txt` is a four byte file containing the word "test", which has a four byte alternate data stream named stream.txt that also contains the word "test".

```
md5sum test.txt → 098f6bcd4621d373cade4e832627b4f6
cksum test.txt → 3076352578
```

After editing the alternate data stream to read "edited" instead of "test":

```
md5sum test.txt → 098f6bcd4621d373cade4e832627b4f6
cksum test.txt → 3076352578
```

[4] "Virus Scanner Inadequacies with NTFS." Dartmouth's Institute for Security Technology Studies.

After removing the alternate data stream from the file:

```
md5sum test.txt → 098f6bcd4621d373cade4e832627b4f6
cksum test.txt → 3076352578
```

These tests show that these file integrity programs do not take the alternate data stream into account when making their calculations. This means that while they do protect the integrity of the contents of the primary, unnamed stream, they do not protect the integrity of alternate data streams. Indeed, these tools must be used in conjunction with an application that identifies the presence of alternate data streams in order to truly verify the integrity of the file and not just the integrity of the primary data stream. It would be trivial to create an implementation of the MD5 and/or CRC algorithms that would use the combination of all the data streams rather than just the primary in calculating the integrity value.

On the other hand, the modification date and time for a file does change if an alternate data stream is added or modified, so this can be a clue to vigilant system administrators.

Alternate data streams cannot be surgically removed without third-party software

Another major difference between regular files or directories and alternate data streams has to do with the operating system support for removing them. The standard delete operations built into Windows Explorer and even the `del` command-line application cannot remove alternate data streams. To eliminate the stream with these utilities, it is necessary to remove the entire file. As mentioned previously, some utilities like Notepad or the `cat` program from the Windows Resource Kit are able to edit alternate data streams. So, they can remove data from the stream, but they cannot remove the data attribute that causes the stream to exist.

Using the DeleteFile function from the Windows API, a programmer can surgically remove the alternate data stream from a file, not disturbing the contents of the unnamed stream. However, Microsoft chose not to implement this very basic functionality in their GUI or command-line tools. One of the shell extensions described later in this document gives the user the ability to utilize the DeleteFile function to remove an alternate data stream.

Streams and exclusive locks

An exclusive lock is a way for a process running on Windows to exclude all other processes from reading or writing to a particular file. Obtained by executing a function in the Windows API called LockFileEx, exclusive locks are rarely used in Windows applications due to the availability of shared read and write locking, which lock portions of a file but not the entire file. Nevertheless,

exclusive locks are necessary for certain applications that need to insure the integrity of a complete file for a running process.

Each stream in a file has separate lock attributes, but very few Windows applications distinguish between the locks on unnamed and named streams. So, a program unaware of alternate streams and locks only looks at the primary stream to see whether a file can be opened, while an application that understands alternate data streams will check the lock on the stream being referenced, primary or alternate. Furthermore, in order to enumerate the named streams in a file, the Windows API function CreateFile must be called to open with the correct arguments to open the primary stream. If the correct arguments are not used and the primary stream has been locked by a running process, the function fails and the named streams cannot be enumerated. All alternate data stream scanners tested later in this paper fail to use the appropriate arguments to CreateFile and are unable open the primary stream before proceeding with enumeration. Therefore, they do not work for scanning exclusively locked files.

Unfortunately, the following scenario could occur:

1. A user could find a file with that has been exclusively locked by a system process that they have permission to write to.
2. They can then use a stream-aware application like Notepad to add data to a named stream. For extra security, the user will give the stream a very complicated name.
3. Alternate data stream scanners will not be able to find the named stream on the locked file. Furthermore, because the primary stream is locked, the file cannot be deleted, moved, or renamed as long as the process is running.
4. The user will be able to read and write data from the alternate stream using the complicated name that he or she selected, even while the primary stream remains locked, evading detection of scanners.

To enumerate the streams in the file it would be necessary to shut down the process that is keeping the file locked. If the process does not terminate cleanly, the lock will remain for a certain period of time but will eventually be cleared. At this point, the alternate data stream scanner will be able to find the name of the stream and it can be removed.

To help protect against this kind of exploit, it is necessary to maintain permissions on files that are locked by system processes to prevent users from writing to them. File permissions are set on the entire file, not each individual stream, so read-only permissions guarantee the inability to write an alternate stream. Also, it is important to ensure that scanning for alternate data streams is done after terminating any non-system processes that would keep locks open. Another solution to this problem would be the modification of the alternate stream scanner to use CreateFile in such a way that a file handle is created but the file is not technically opened for reading. The alternate data stream shell extensions

described later in this paper use CreateFile appropriately and are **not** subject to the limitations described above.

How can alternate data streams be used maliciously?

While not inherently dangerous, the different ways that alternate data streams are handled by Windows as compared to primary streams can make them vulnerable to exploit by malicious users. Alternate streams can be used to hide data, store executable code, or even perform denial of service attacks.

Data can be hidden in files or directories

Given the inability of most Windows applications to enumerate the alternate data streams in a file or directory, these streams make excellent locations to hide malicious data. This vulnerability becomes very severe when compounded by the problem of exclusive locking discussed previously that may prevent these streams from being enumerated at all by most scanning applications. Alternatively, depending on the amount of data that needs to be hidden, the malicious user could use an alternate stream with a name that would not be suspicious. A good example of this would be the small "SummaryInformation" stream that contains summary information displayed in the preferences tab and is common on certain Windows documents. A forensic analyst would have to specifically search all of the hidden streams on the volume to be confident that no data was hidden there, and they could not determine the content of the file from the stream name alone.

What kind of malicious data can be hidden in an alternate data stream? Any data that can be stored in a primary unnamed stream can be stored in a hidden named stream as well. To add and edit the data in an alternate data stream, a user only needs a stream-aware tool like Notepad or `cat`. Alternate data streams can provide a covert channel for the storage of stolen intellectual property, like a formula for a new pharmaceutical or an archive of source code to a proprietary operating system. Potentially even more dangerous is the ability for executable code to be stored in an alternate data stream, creating a new vector for malicious software, also called malware.

Code can be executed directly from an alternate data stream

Unfortunately, data can not only be stored in alternate data streams, but also executed directly from it. In NT4, alternate stream data could be executed using the "start" command supplied with the operating system; however, according to Kaspersky and Zenkin[5], there are at least five ways to execute different types of data in Windows 2000. They created three alternate streams in a copy of the Notepad executable, one stream has an exact copy of Calculator, the second contains a test Visual Basic script that creates a message box containing the words "Hello World!", and the third stores a .cmd file that echoes "Hello World!" to the command-line. They found that each of the following five

[5] Kaspersky, Eugene and Zenkin, Davis. "NTFS Alternate Data Streams"

methods would cause at least one of the three file types to successfully execute on Windows 2000:

1. Executing the stream from the Run window as `file:\\notepad.exe:<stream name>` works for the .exe stream and the .vbs stream
2. Executing the Visual Basic script from the command line using the Windows Scripting Host by running `wscript notepad.exe:<VB stream name>`
3. Creating a shortcut to `notepad.exe:<stream name>` will execute both the .exe and .vbs streams
4. Placing a shortcut to the stream in the Windows Startup folder will cause the .exe and .vbs streams to be executed when a user logs in.
5. Adding a test key with value "`notepad.exe <stream name>`" to HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run will cause .exe and .vbs streams to be executed on system startup.

Antivirus software vendors have claimed that they are not concerned about searching for virus data stored in an alternate stream because it could not be executed without a "starter" in the primary stream. The most important result of the experiments conducted by Kaspersky and Zenkin was determining that alternate streams can be executed directly, and they do not need any sort of starter code in the primary stream. This fact makes alternate streams a very viable and covert vector for many types of malware including viruses and Trojan horses.

Denial of Service attacks

Alternate streams also provide a very effective means to conduct a denial of service attack on a host. Because most Windows applications cannot show the size of any alternate streams on a file, a malicious user could easily hide an extremely large file in a named stream so that another user could not detect its presence. This ability to hide large amounts of data makes it almost trivial to fill an NTFS volume to capacity, ultimately rendering it unable to hold any additional data. A completely full system volume almost always causes severe OS and application instability. At this point, without a third-party alternate data stream scanner, a system administrator would be unable to track down the file that contained the oversized stream. The severity of the problem can escalate significantly for a stream placed on a file exclusively locked by the system, because as mentioned previously, popular scanning applications cannot find a named stream on an exclusively locked file. If a process vital to system operation maintains the lock and it cannot be shut down, the only way to find the stream currently is to mount and scan the volume on a separate machine.

What tools are available to work with alternate data streams?

A variety of tools available from Microsoft and third-party vendors will find or manipulate alternate streams.

Microsoft tools

Microsoft distributes tools that can be used for manipulating, but not finding or removing, alternate data streams with their Windows operating system and with the Windows Resource Kit. Both the GUI tool Notepad and the command-line tool `cat` (which comes with the Resource Kit) are very useful for creating new streams or editing existing streams, provided the user knows the name of the stream they wish to edit.

Notepad

When editing alternate data streams with Notepad, the syntax is:

```
notepad <filename>:<stream>
```

This command opens the alternate data stream named `<stream>` on a file named `<filename>` in an editing window just like it would for a primary data stream. If the stream does not exist, a message box will prompt the user to create a new "file" with the syntax above. If the stream already exists, it will be displayed in the window normally. The Notepad File→Save operation will save the edited text into the stream.

cat

While Notepad works best for editing ASCII text, the cat command combined with command-line redirection, also known as the > and < operators, provides the user with the ability to redirect files of any type into and out of an alternate data stream. To place a file into an alternate data stream using cat, the syntax is:

```
cat <input file> > <filename>:<stream>
```

This command will place the file `<input file>` into the alternate data stream named `<stream>` on a file named `<filename>`. If the stream already exists, it will be overwritten, and if it does not exist it will be created. The input file can be a text file or a binary file. Indeed, using `cat` in the command above would be the best way to store executable data in an alternate stream. Conversely, to copy a stream out of a file, use the command:

```
cat <filename>:<stream> > <output file>
```

The alternate data stream will be copied into the file named `<output file>`. This command does not delete the alternate data attribute from the original file; therefore, it only works to copy, not move, a stream to a separate file.

Third-Party scanners

A number of third-party alternate data stream scanners have been available for years, including LADS, Streams, CrucialADS, and a product called TDS-3. Each of these scanners has its own strengths and weaknesses and they will be evaluated on their ability to find alternate data streams, speed, and usability. For the first test, three files and directory pairs – one normal, one encrypted, and one compressed – were created and each given the same alternate data stream. The second test will compare the applications' speed in scanning an entire hard drive for alternate streams. The final evaluation is a subjective analysis of each utility's usability for scanning and analyzing alternate data streams on a volume.

LADS

The most popular alternate data stream scanner is Frank Heyne's List Alternate Data Streams (LADS). LADS is a freeware command-line application and can be downloaded as a binary only from Frank's site at http://www.heysoft.de.

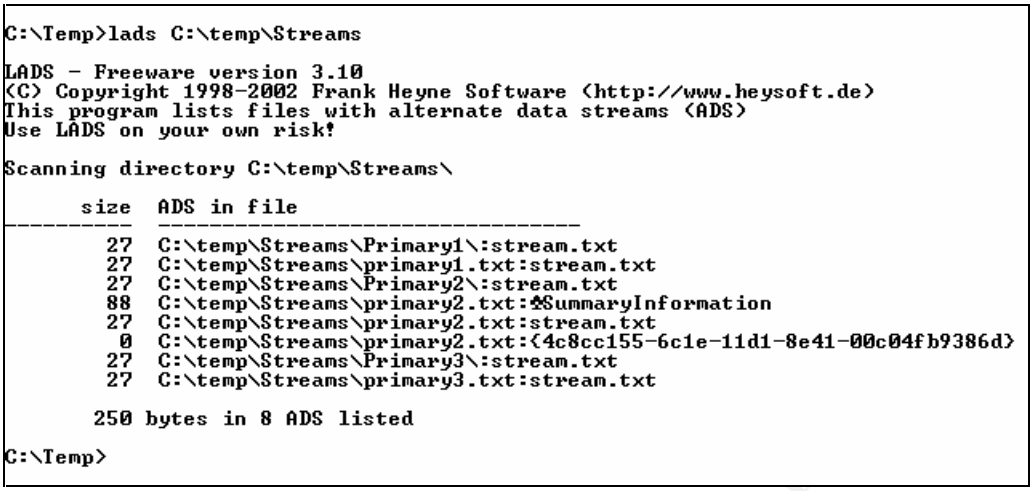
Usage

The command-line usage for LADS is:

```
lads [Directory] [/S] [/D] [/A] [/Xname]
```

`Directory` is the directory to scan, but if it is left blank LADS will scan the current directory. LADS has a few command-line switches that can be set: `/S` to recursively scan subdirectories, `/D` to put LADS in debug mode, `/A` to sum all of the bytes found in hidden streams, and `/Xname` to exclude alternate data streams with a particular name. The output of LADS during a normal alternate data stream scan appears in Figure 5:

Figure 5: A screenshot of LADS



Evaluation

LADS passed test one with flying colors; it found all six alternate data streams in the normal, encrypted, and compressed files and folders. In test two, LADS also performed rather well, taking only 18.5 minutes to scan 9GB of files and directories for alternate data streams.

Though it finds all alternate streams and is reasonably quick, LADS is lacking in several respects with regard to usability. First, it only processes absolute (i.e. `C:\directory`) directories as options and does not accept relative (i.e. `..\directory`) directories as valid. This makes it difficult to use LADS to scan deeply nested directories because the user must type the entire path to the directory being searched. In addition, LADS lacks decent reporting capability. No option outputs the results to a file for easy analysis. If the volume is large and has many alternate data streams (or open files, which print an error), the output will be pages long. The output can be redirected to a file using the command-line redirector, `>`, but the format of the output is not conducive to scripted analysis. It would be nice if LADS had a switch to output to a comma-separated value (CSV) file that could be viewed in Excel or easily processed by a Perl script.

This application is also only an ADS scanner, it does not give the user the ability to view, save, or remove data in the streams themselves, though it does report the size of the stream in its output. Also, LADS does not support the scanning of a single file, which complicates situations in which a user wants to find alternate streams on a file in a very large directory. LADS is vulnerable to the exclusive file locking exploit discussed previously.

Streams

Streams is a small, command-line based alternate data stream scanning tool from Sysinternals. It is freeware, available in both binary and source-code

form, and can be downloaded directly from Sysinternals at http://www.sysinternals.com/ntw2k/source/misc.shtml.

Usage

The command-line usage for Streams is:

```
Streams [-s] <file or directory>
```

Streams is able to scan a single file, scan a single directory, or recursively scan files and directories with the `-s` switch. The output of LADS during an alternate data stream scan is shown in Figure 6:

Figure 6: A screenshot of Streams

```
C:\Temp>streams -s streams

Streams v1.3 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2001 Mark Russinovich
Sysinternals - www.sysinternals.com

C:\Temp\streams\Primary1:
      :stream.txt:$DATA 27
C:\Temp\streams\primary1.txt:
      :stream.txt:$DATA 27
C:\Temp\streams\Primary2:
      :stream.txt:$DATA 27
C:\Temp\streams\primary2.txt:
   :SummaryInformation:$DATA   88
      :stream.txt:$DATA 27
   :{4c8cc155-6c1e-11d1-8e41-00c04fb9386d}:$DATA        0
C:\Temp\streams\Primary3:
      :stream.txt:$DATA 27
C:\Temp\streams\primary3.txt:
      :stream.txt:$DATA 27

C:\Temp>
```

Evaluation

Streams successfully passed test one, finding all of the hidden streams in the six test files. It also performed well in the speed test, taking only 18 minutes to scan 9GB of files and directories.

Streams, like LADS, is of limited usability. While it can handle absolute and relative paths and is able to scan a single file for alternate streams, Streams lacks a way to generate output that can be easily analyzed. Like LADS, Streams is a command-line utility that prints out a list of alternate data streams in a non-standard format. It does not offer any switches to give the user the ability to request output as a CSV file or other standard format. For systems that may have hundreds of alternate data streams, this makes analysis of the information much more difficult.

Also, like LADS, Streams can only scan for alternate data streams; it does not give the user the ability to view, save, or remove an alternate data stream from a file or directory. Streams is also vulnerable to the exclusive file locking exploit.

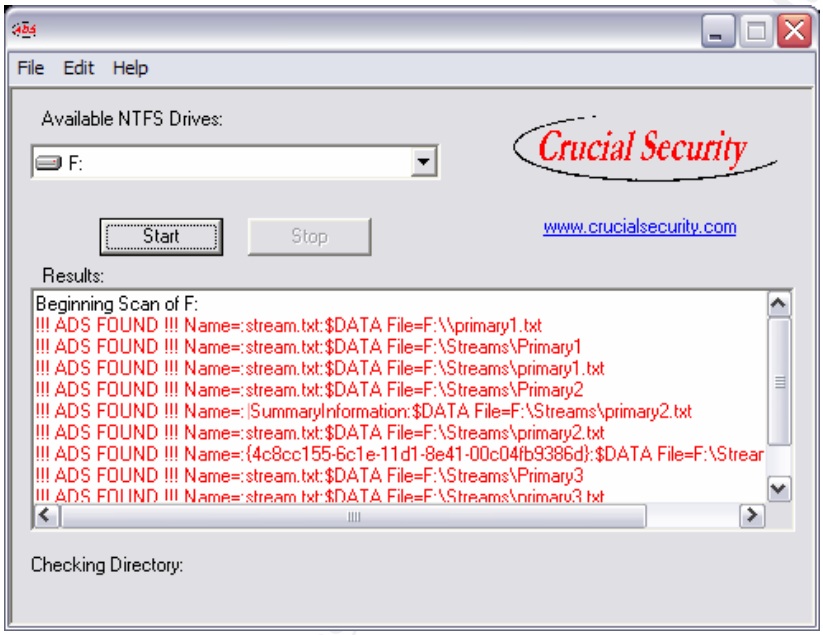
CrucialADS

CrucialADS is a freeware alternate data stream scanning utility with a graphical user interface. It is available as a binary only from Crucial Security at http://www.crucialsecurity.com.

Usage

CrucialADS is a very easy to use program. Executing it produces a window that lets the user select the drive to scan. The user can then control the scan with the Start and Stop buttons. The results appear in a window with files or directories that have alternate data streams highlighted in red. Also, there is an option in the File menu to save the results of the scan to a file. Figure 7 shows CrucialADS's interface:

Figure 7: A screenshot of CrucialADS



Evaluation

CrucialADS found all six alternate streams in the test files and directories, receiving perfect marks for thoroughness. In addition, as an ADS scanner, CrucialADS performed the most quickly, scanning the 9GB volume in a mere 16 minutes and 15 seconds.

CrucialADS's graphical user interface makes it easy to use if a user is doing a very specific kind of scanning. Unfortunately, does not provide many options, only giving the user the option of scanning an entire volume. This makes it unacceptable as a general-purpose alternate data stream scanner. It is not appropriate to force a user to scan an entire volume if they merely want to search for an alternate data stream on a particular file; while scanning a single file may take only a second, a full volume scan with CrucialADS takes much, much longer.

CrucialADS is the first alternate stream scanner evaluated that gives the user the option of saving the results of the scan to a file. Unfortunately, the format of the file looks identical to the format of the data on the screen – very unhelpful from an analysis standpoint. Like the other two products evaluated so far, hundreds of data streams will produce pages of output, and CrucialADS does not export the data in an easily manipulated format.

In addition, like LADS and Streams, CrucialADS does not give the user the ability to view, save, or remove an alternate data stream from a file or directory, only to search for them. CrucialADS is also vulnerable to the exclusive lock exploit.

Trojan Defense Suite

Trojan Defense Suite (TDS-3) is a shareware Trojan scanning application that contains an alternate data stream scanner. It is available as a binary from Diamond Computer Systems at http://tds.diamondcs.com.au/.

Usage

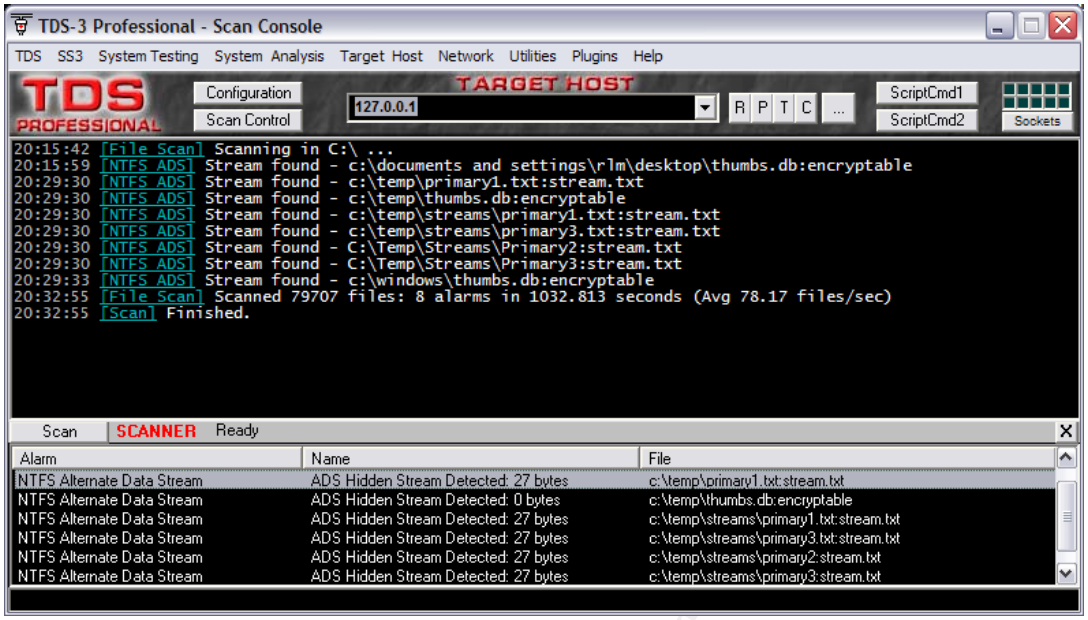
TDS-3 is a full-featured Trojan scanning application with a very broad feature set. This evaluation, however, will focus specifically on its alternate data stream scanning capability.

An alternate stream scan can be set up through the System Testing → Scan Control menu. Select the drive, directory, or file to be scanned from the left side of the scan list and move it to the right. There are two options, "Scan NTFS ADS Hidden Streams" and "Show all NTFS ADS Streams" that must be checked to enable alternate data stream scanning.

In the ADS option settings, several options control the scan. "Ignore non-executable streams" will ignore any streams that do not contain executable data, which could be useful if a user were searching for Trojan or virus data that has been hidden in a stream. The next option "Ignore streams smaller than" will cause the scan to skip over smaller files. This may be useful in some instances (setting the size to 1 to eliminate 0 byte streams), but a user should not think that streams below a certain size are not dangerous. Data can be fragmented among many streams, making it much more difficult to find if this option is enabled. The final option in that window "Ignore streams containing keyword(s)", could be used to ignore the little "SummaryInformation" stream that is generated by the Summary tab in Windows Explorer. However, just because the stream is named "SummaryInformation" does not mean that it actually contains summary information, so this option should be used carefully.

Once a set of streams has been found by TDS-3, the user can perform a variety of activities on the stream. Right-clicking a stream brings up a menu of options that let the user view the stream in Notepad, save the stream to a separate file, delete the stream, or view stream properties. The stream properties window includes the name and size of the primary stream as well as the name and size of the selected alternate stream. Trojan Defense Suite's user interface can be seen in Figure 8:

Figure 8: A screenshot of TDS-3



Evaluation

Trojan Defense Suite was the only application to fail the first portion of the test. Scanning the test files revealed only four, not six, alternate data streams. TDS-3 was unable to see the alternate stream in the normal directory or in the encrypted file. This deficiency makes TDS-3 a very poor scanning product. Hiding an alternate data stream in a directory is not an uncommon thing, so it seems very strange that this software would not be able to detect it. On the other hand, TDS-3 completed the scan of the 9GB dataset in 17 minutes and 12 seconds, making it the second fastest scanner.

Though the interface is a little cluttered and confusing, TDS-3 otherwise gets high marks for usability. It allows a user to select a volume, directory, or file for scanning, and even provides the ability to scan multiple unique files in different directories with a single scan. When it can find an alternate stream on a file or directory, it gives the user the option to view it, save it to a separate file, or remove it. TDS-3 is the only scanning application with this functionality. One of the few limitations of Trojan Defense Suite is its inability to save the scan in a format that can be shared among other applications and scripting languages.

Because TDS-3 did not find all of the alternate streams in the test, it is not an appropriate product to use to search for malicious data hidden in named streams, but may be sufficient for other purposes. Trojan Defense Suite also fails to find streams in files that have been exclusively locked.

Beyond scanning: Integrating alternate data stream information into Windows

The biggest problem with alternate data streams is their lack of visibility in the operating system. They are supported by some applications and not by others, but there is no piece of software shipped with Windows that gives the user the ability to see that an alternate data stream exists on a file without knowledge of the stream name. A number of third-party solutions find alternate data streams and in some cases allow them to be manipulated; however, the lack of this functionality in the operating system gives malicious users the edge. An administrator will have to know that alternate data streams exist and will have to use a third-party tool specifically designed to search for them to bring them out "into the open". The remainder of this paper will discuss a set of Windows shell extensions that integrate alternate data streams into the typical user environment. By doing this, alternate streams will no longer be a "trick" or something that is scanned for, but rather a part of the file system that the user becomes accustomed to seeing, giving them the edge in identifying irregularities the usage of these streams. These extensions also use the Windows API in a way that lets them scan files and directories that have been exclusively locked, giving them a unique edge.

No tool is perfect, so it is essential to understand the limitations of the shell extensions as they are implemented. After a discussion of these limitations, there will be an introduction to the few function calls necessary to enumerate and access data from alternate streams in the API and then an in-depth walkthrough of each extension.

Limitations of the shell extensions

Shell extensions are inherently limited by the power of the user running the shell. To that end, these extensions are not able to bypass Windows security and find alternate data streams on files that the user does not have read permission on. It follows that these streams can also not be deleted if the user does not have access to write to the file. In addition, these particular extensions use API calls that are specific to Windows NT-based systems only, so they will not work on Windows 95/98, only on NT, 2000, and XP.

How do you use the Windows API to gather information about alternate streams?

A certain collection of functions in the Windows API is capable of understanding alternate data streams and should be used when writing an application that intends on handling them.

CreateFile

According to Microsoft's MSDN library, "The CreateFile function creates or opens a file, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, or pipe. The function returns a handle that can be used to access the object." In layman's terms, this means that CreateFile is the function that you use to access data. A programmer tells the function what he wants to access and how and it gives him back a "handle" that can be used to read, write, or do dozens of other operations with the data. For the purpose of accessing alternate data streams, it is very important to note that the CreateFile function does understand alternate stream syntax. It will open a handle to a file or directory using the "<file or directory name>:<stream name>" naming convention. Therefore, it can be used to access the alternate data stream information programmatically, and the handle for an alternate stream can be passed to any other function just as the handle for a primary stream would be.

The exact syntax of the CreateFile command is available in the MSDN Library, but because it will be relevant later the basic usage will be outlined. CreateFile accepts a set of arguments including the file or directory name to be created or opened, the action to be taken (create or open), and the sharing mode of the access. There are other arguments that can be used to control other aspects of the open, but they are not relevant to this paper. For example, look at the following function call:

```
HANDLE hfile = CreateFile((LPCTSTR) "test.txt:stream.txt",
                          GENERIC_READ, FILE_SHARE_READ |
                          FILE_SHARE_WRITE, NULL,
                          OPEN_EXISTING, NULL, NULL);
```

In this case the function opens the alternate stream named `stream.txt` on the file named `test.txt` for reading (`GENERIC_READ`). The file is opened allowing read sharing and write sharing (`FILE_SHARE_READ | FILE_SHARE_WRITE`), and there is a flag (`OPEN_EXISTING`) to indicate that the function should only open the file only if it already exists, rather than creating it and then opening it. The function returns the file handle into the variable `hfile`, which can then be passed to another function. This is just one example of the many uses of the CreateFile function, several of which will be seen in the source code walkthrough.

BackupRead and BackupSeek

The documented way to enumerate the alternate data streams given a file handle is to use the BackupRead and BackupSeek functions. Microsoft provides them for backup programs to use when reading whole files to tape or to another backup device. These functions can, however, be exploited scan a file for alternate data stream headers. Think of them as manipulating a finger that points to a word in a line of text. BackupRead reads the word (byte) under the finger

(pointer), while BackupSeek moves the finger ahead a certain number of words (bytes) without reading. Without going into the details of the code, they work to read alternate data stream names by doing the following:

1. Use BackupRead to read the header of the stream, which tells us the length of the data that follows (D) and the length of the name of the stream (N), if it has a name.
2. The name information always immediately follows the header, so use BackupRead to read N bytes of data. This is the name, so store it in a variable.
3. Use BackupSeek to skip over D bytes of data until you reach the next header, if there is one. Start back over at one with the pointer at the start of the next stream header.

By proceeding in this fashion, a program can eventually read all of the names of all of the streams in a file or directory. Unfortunately, this function requires that the file be opened for reading, so it cannot be used on files that have been exclusively locked. Also, because it involves moving a pointer all the way through a file, it could be very slow, depending on the length of the file being accessed. However, there is another way!

NTQueryInformationFile

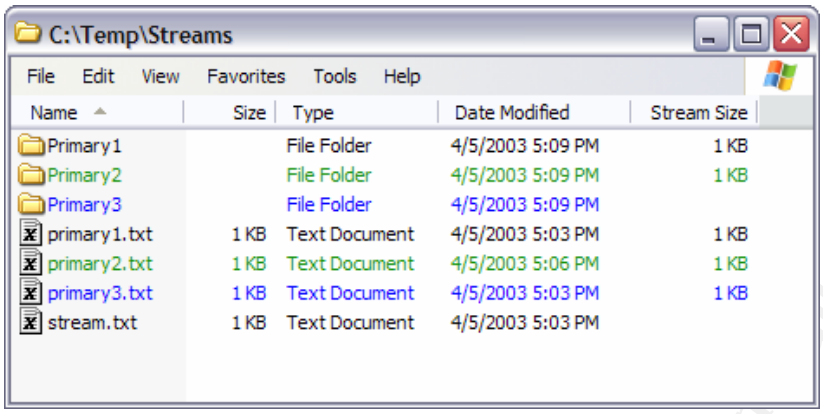
NTQueryInformationFile is the best function to use to find the names of an alternate data stream in a file or directory. It is a very fast function, and it does not require opening a read handle to obtain attribute information. This means that NTQueryInformationFile can be used on files that are exclusively locked! Unfortunately, NTQueryInformationFile is hardly documented at all. It resides in the NTDLL.DLL file that ships with all versions of Windows NT/2000/XP, but there is limited information about it on the MSDN Library. Nevertheless, information about the data structures that the function uses is available from various sources online. Essentially, it takes a file handle and a file information structure as arguments and writes the requested information into a file stream information buffer. This is much more efficient than the loop used by the BackupRead-BackupSeek method because it is implemented directly by the system-level function and the data can be easily read out of the file stream structure by the programmer.

Creating a column handler to show the size of alternate data streams in detail view

Column handlers are shell extensions that allow certain information about a file or directory to be displayed in a column in the detail view of Windows Explorer. Default column handlers include name, size, type, and date modified, though there are many other column handlers supplied with Windows. The column handler is an excellent place to show alternate data stream information

because it can be read at first glance and the view can even be sorted by the new size column. The column handler shell extension will produce the column illustrated in Figure 9.

Figure 9: A folder with several files and folders and a stream size column



Code walkthrough

A column handler is a column provider object that exports the IColumnProvider interface. IColumnProvider is a way for a programmer to interact with the Windows Shell and create their own columns. The IColumnProvider interface has three member functions:

Initialize – initializes the IColumnProvider interface
GetColumnInfo – requests information about a particular column
GetItemData – requests information about a particular file

Initialize

So to create a new column, an initialize function must be defined. In the case of this column, there is no initialization that needs to be done, so the Initialize function returns S_OK, which means “initialization done”.

GetColumnInfo

The GetColumnInfo provides information to the shell about how the column will be laid out, what kind of data is kept in it, and what it’s name and description is. Here is the source code for the GetColumnInfo function:

```
HRESULT CStreamColInfo::GetColumnInfo(DWORD dwIndex,
                                      SHCOLUMNINFO *psci)
{
  if ( dwIndex > 0 )
    return S_FALSE;
```

The function takes an index as an argument, if a single IColumnProvider interface is providing multiple columns, `dwIndex` will identify each column, with

column 1 being `dwIndex` 0, column 2 having `dwIndex` 1, etc. This interface only provides one column, so return false if the shell asks for more than that.

```
  // Use the object's CLSID
  psci->scid.fmtid = *_Module.pguidVer;
  psci->scid.pid = 0;
  // The data is a string
  psci->vt = VT_LPSTR;
  // The column will be right aligned
  psci->fmt = LVCFMT_RIGHT;
  // The column is a string and may return slowly
  psci->csFlags = SHCOLSTATE_TYPE_STR|SHCOLSTATE_SLOW;
  // Default width in characters
  psci->cChars = STREAMCH_DEFWIDTH;
```

This code sets up the characteristics of the column. A unique identifier is assigned to the column, it is set to hold string data, and it is right aligned like the default size column. Then, the next statement indicates that the column will store strings and that it may return slowly. Finally, the default width of the column is set.

```
  // Caption and description
  wcsncpy(psci->wszTitle, L"Stream Size", MAX_COLUMN_NAME_LEN);
  wcsncpy(psci->wszDescription,
          L"Provides the total size of the streams in the object",
          MAX_COLUMN_DESC_LEN);

  return S_OK;
}
```

These functions copy the string “Stream Size” into the title of the column and the string “Provides the total size of the streams in the object” into the description, truncating them each if necessary. The function then returns S_OK to indicate that the column information has been set up.

GetItemData

GetItemData is called for each file or directory in the detail view. It decides whether to display a value in the column or not and if so, what value to display. This function is the part of the code that determines whether we need to enumerate the alternate data streams on the file or directory. Here is the source code:

```
HRESULT CStreamColInfo::GetItemData(LPCSHCOLUMNID pscid,
                                    LPCSHCOLUMNDATA pscd,
                                    VARIANT *pvarData)
{
  USES_CONVERSION;
  // Convert filename to other string types
  LPCTSTR szFilename = OLE2CT(pscd->wszFile);
  LPTSTR szFilenameNTFS = OLE2T(pscd->wszFile);
```

Convert the filename of the file or directory, stored in `pscd->wszFile` into other formats so it can be used later.

```
  // Check if the object is stored on an NFTS volume
  if (! IsNTFS(szFilenameNTFS))
    return S_FALSE;

  // Do not look for streams in an offline file
  if ( pscd->dwFileAttributes & FILE_ATTRIBUTE_OFFLINE )
    return S_FALSE;
```

The first statement checks to make sure that this file or directory is actually on an NTFS volume, because alternate data streams only exist in NTFS. The next statement returns false if the file or directory is offline. An offline file is not stored on disk, so it does not need to be checked for alternate streams.

```
 TCHAR szBuf[STREAMCH_MAXSIZE];
  ZeroMemory(szBuf, STREAMCH_MAXSIZE);

  // Get the size of the stream and put it in szBuf
  if (GetStreamSize(szFilename, szBuf) == S_OK) {
    CComVariant cv(szBuf);
    cv.Detach(pvarData);
    return S_OK;
  } else {
    return S_FALSE;
  }
}
```

Several storage variables are created and then the GetStreamSize function is called to find the size of the streams in a particular file or directory. This is the workhorse function of the column handler implementation. If it successfully returns a size, the size is stored in `szBuf` as the column data; otherwise, the column is left blank.

GetStreamSize

GetStreamSize is the primary function of the column handler. It uses the information from NTQueryInformationFile to enumerate the streams, summing up the sizes of every named stream and returning a total file size in kilobytes. Here is the source code for GetStreamSize:

```
HRESULT CStreamColInfo::GetStreamSize(LPCTSTR szFileName,
                                      LPTSTR p)
{
  USES_CONVERSION;

  // Retrieve NtQueryInformationFile function location
  // from NTDLL.DLL
```

```
static class ListFileContext {
  private:
    HINSTANCE hDll;
  public:
    NtQueryInformationFileFunc *pQIF;
    ListFileContext() {
      hDll = LoadLibrary ("NTDLL.DLL");
      pQIF = (NtQueryInformationFileFunc *)
        GetProcAddress (hDll, "NtQueryInformationFile");
    }
    ~ListFileContext() {
      FreeLibrary (hDll);
    }
} Context;
```

This class declaration grabs the location of the NTQueryInformationFile function from NTDLL.DLL and stores it internally for later use.

```
// Setup variables for NtQueryinformation File
IO_STATUS_BLOCK Iosb;
unsigned char Buffer[2048];
FILE_INFORMATION_CLASS eClass;
FILE_STREAM_INFORMATION *pFsi;

// Open a file handle, with 0 for desiredAccess to
// bypass exclusivity checking
HANDLE hFile = CreateFile( szFileName, 0,
  FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
  OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, 0 );
// If the file handle is invalid, return (unknown)
if (hFile == INVALID_HANDLE_VALUE) {
  wsprintf(p, _T("(unknown)"));
  return S_OK;
}
```

The first block of code declares a set of variables that will be used later to retrieve stream names and sizes. The second block of code opens a file handle using the value “0” for the second argument. This allows CreateFile to open a handle on an object that has been exclusively locked, giving this column handler the ability to see streams that have been hidden on locked files. The last statement returns the value “(unknown)” which will be displayed in the column in the instance that a handle cannot be opened. This will happen when a user does not have permission to read the file or if the file is for any other reason unreadable. Also, the flag `FILE_FLAG_BACKUP_SEMANTICS` must be set to allow CreateFile to return the handle of a directory.

```
// Get stream information into Buffer
eClass = FileStreamInformation;
NTSTATUS rc = (Context.pQIF) (hFile, &Iosb,
  (PVOID) Buffer, sizeof(Buffer), eClass);
```

This statement calls NTQueryInformationFile on the handle named `hFile`, returning the stream information into a structure in `Buffer`.

```
// If there is no information in the status block,
// we are looking at a directory with no stream
if (! Iosb.Information ) {
  CloseHandle( hFile );
  return S_FALSE;
}
```

If the handle is to a directory with no stream, the `Iosb.Information` variable will be NULL, so the handler can immediately return false and save computing cycles.

```
// Setup variables for stream summing loop
WCHAR wszStreamName[MAX_PATH];
LARGE_INTEGER totalStreamSize;
totalStreamSize.QuadPart = 0;

// For each stream in the file
for (pFsi = (FILE_STREAM_INFORMATION *) Buffer;
  1;
  pFsi = (FILE_STREAM_INFORMATION *)
  ((BYTE *) pFsi + pFsi->NextEntryOffset) {
```

The first block of code sets up the variables that will be used in this summing loop. The second iterates through the streams, putting the stream information structure into the `pFsi` variable.

```
  // Pull the stream name out so we can make sure
  // that it's a named stream that we're summing
  ZeroMemory(wszStreamName, MAX_PATH);
  memcpy( wszStreamName,
    pFsi->StreamName,
    pFsi->StreamNameLength );
  wszStreamName[ pFsi->StreamNameLength/2 ] = 0;
```

The handler grabs the name of the stream from the stream information structure and copies it into `wszStreamName` so it can be checked later.

```
  // Is it a named stream?
  if( wcsicmp( wszStreamName, L"::$DATA" )) {
    totalStreamSize.QuadPart =
    totalStreamSize.QuadPart + pFsi->StreamSize.QuadPart;
  }
```

This statement checks to see that the stream we are looking at is not the default, unnamed stream (“::$DATA”). The handler should only sum the sizes of the alternate streams. If the stream is named, the handler increments the

`totalStreamSize` variable by the size of the stream (`pFsi->StreamSize.QuadPart`).

```
    // Move to the next stream
    if (! pFsi->NextEntryOffset)
      break;
  }
  CloseHandle( hFile );
```

Continue iterating through the streams as long as there are streams to iterate through. When there are no more streams, break out of the loop and then close the file handle that was opened.

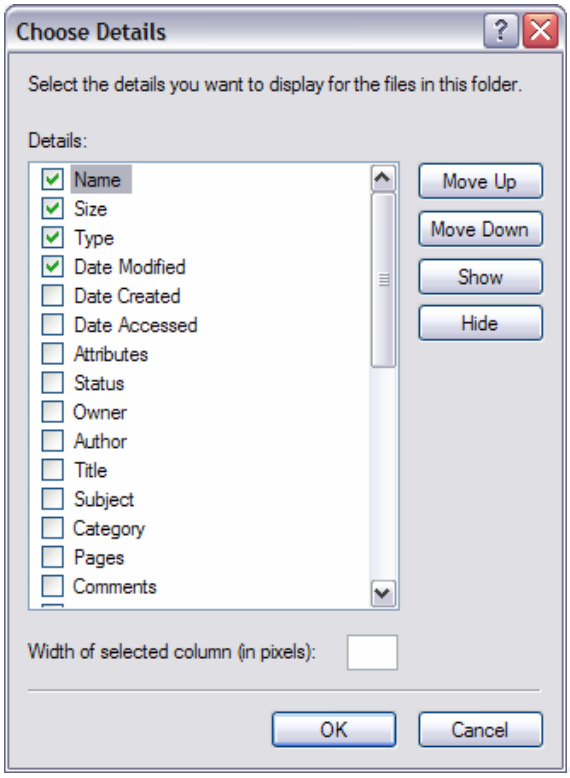
```
  if (totalStreamSize.QuadPart > 0) {
    // If the file is smaller than 1KB, make it 1KB
    if (totalStreamSize.QuadPart < 1024)
      totalStreamSize.QuadPart = 1024;
    wsprintf(p, _T("%s KB"),
      Commify((unsigned int) totalStreamSize.QuadPart / 1024));
  } else {
    return S_FALSE;
  }
  return S_OK;
}
```

This block of code checks to make sure there is a file size to display, makes sure that the minimum file size is 1KB, and formats the size string, writing it to `p`. The Commify function takes an integer and places commas in every three digits out so that it looks nice. The GetItemData function reads the variable `p` and inserts it into the column. If GetStreamSize returns false at any point, GetItemData does not print anything in the Stream Size column.

Usage

The column handler is very easy to install and use. After installing the package by running `Setup.msi`, the additional column will appear in the list of more columns in Windows Explorer. Right click on the column headers at the top of Detail view and select the option at the bottom of the list labeled “More…” and the window in Figure 10 will appear.

Figure 10: The Choose Details window

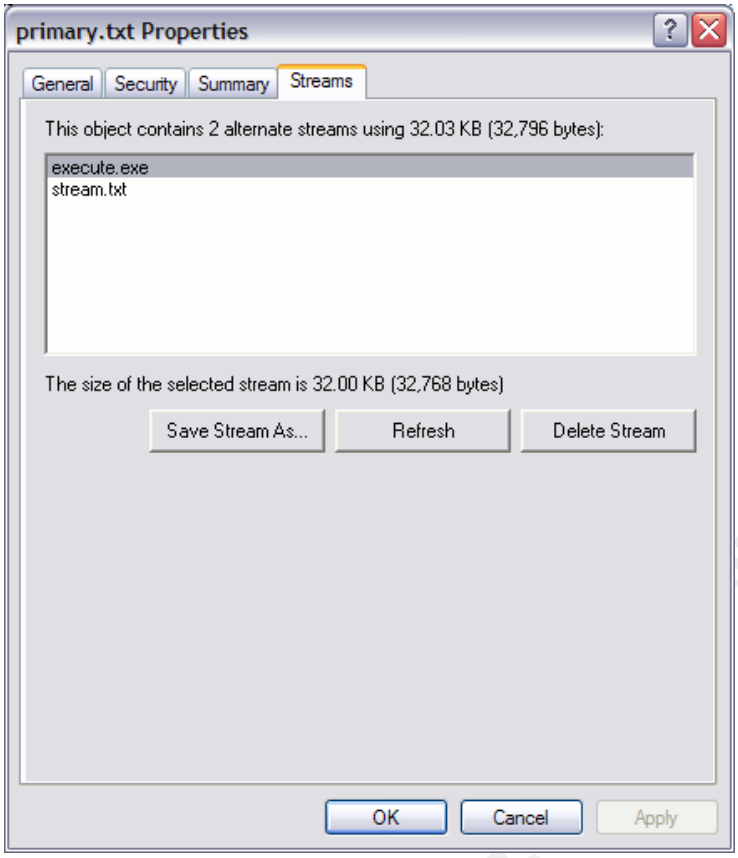


In this list will be a new column called “Stream Size”. Checking the box for the new column and clicking OK will cause it to appear in the Details view of Windows Explorer. The column can be moved around with drag and drop or it can be resized on its edges. In addition, clicking the column once will cause it to sort ascending and clicking it again will cause it to sort descending. Even sorted, folders and files remain separate, so the sorting will occur internally to each group as illustrated on Figure 9. Also, it is important to note that if the file cannot be read for some reason, the column handler will display ”(unknown)”. The cause of this message is most likely a permissions conflict. Changing the file permissions so the column handler user has read access, if appropriate, will solve the problem.

Creating an additional property page to view alternate stream names and sizes and perform tasks on them

Property pages are displayed on files or directories when a user right clicks on the icon and selects “Properties”. A tab under the properties screen is the perfect place to store information about the streams in the file because it is consistent and easily accessed. From this property page, the user should be able to see the names of the alternate data streams in the file and their sizes. Additionally, the user can save an alternate data stream to another file or remove it from the record without removing any other data. The property page shell extension will display the page illustrated in Figure 11.

Figure 11: The alternate stream data property page



Code Walkthrough

A property page handler is a property sheet provider object that exports the IShellPropSheetExt interface. The alternate data stream property page implementation only uses the AddPages method of this interface, which creates a property page using a property page data structure.

AddPages

The AddPages method sets up the graphical layout of the page, sets the title, and provides a pointer to the function that will accept input (mouse clicks) from the page. Here is the source code for the AddPages function:

```
HRESULT CEnumStreams::AddPages(LPFNADDPROPSHEETPAGE lpfnAddPage,
                               LPARAM lParam)
{
  PROPSHEETPAGE psp;
  HPROPSHEETPAGE hPage;

  ZeroMemory(&psp, sizeof(PROPSHEETPAGE));
  psp.dwSize = sizeof(PROPSHEETPAGE);
  psp.dwFlags = PSP_USETITLE|PSP_DEFAULT;
  psp.hInstance = _Module.GetModuleInstance();
```

The code block above sets up the property page structure by clearing memory space for it, setting its size, indicating that the property page will have a default layout and use the `psp.pszTitle` variable as the name of the tab.

```
  psp.pszTemplate = MAKEINTRESOURCE(IDD_ENUMSTREAMS);
  psp.pszTitle = _T("Streams");
```

The template for the layout of the page is specified in `IDD_ENUMSTREAMS`, and the first statement loads that template as a resource. `IDD_ENUMSTREAMS` determines the location of the text, list box, and buttons on the page. The second statement titles the tab of the property page.

```
  psp.pfnDlgProc = (DLGPROC) PropPage_DlgProc;
  psp.lParam = (LPARAM) this;
  hPage = ::CreatePropertySheetPage(&psp);
```

This code identifies the PropPage_DlgProc function as the event processor for the page. So, when a user clicks on objects in the page, these events will be passed along to the aforementioned function for processing.

```
  // add the page to the property sheet
  if (hPage != NULL)
    if (!lpfnAddPage(hPage, lParam))
      ::DestroyPropertySheetPage(hPage);

  return NOERROR;
}
```

The last bit of code adds the page that has been created so far into the list of property pages. If for some reason this is unsuccessful, it destroys what has been created so far to free memory.

PropPage_DlgProc

PropPage_DlgProc is the function that obtains event messages from the user as they click on elements of the page. Here is the source code:

```
BOOL CALLBACK CEnumStreams::PropPage_DlgProc(HWND hwnd,
                                             UINT uiMsg,
                                             WPARAM wParam,
                                             LPARAM lParam)
{
  switch(uiMsg)
```

This switch statement acts as flow control for the function. Switch compares the value of uiMsg to one of the cases below. If the value is equal to `WM_INITDIALOG`, it will execute the first block of code. If the value is equal to `WM_COMMAND`, it will execute the second.

```
 {
   case WM_INITDIALOG:
      RefreshStreams(hwnd);
      return TRUE;
```

The code above is executed when the property page is first displayed. The function executes the RefreshStreams function to populate the list box with the names and sizes of the alternate data streams in the file, or to display an error message if it cannot.

```
    case WM_COMMAND:
      if (LOWORD(wParam) == IDC_DELETE)
        DeleteStream(hwnd);
      if (LOWORD(wParam) == IDC_REFRESH)
        RefreshStreams(hwnd);
      if (LOWORD(wParam) == IDC_SAVESTREAM)
        SaveStream(hwnd);
      if (HIWORD(wParam) == LBN_SELCHANGE)
        SelectionChanged(hwnd);
      break;
  }
  return FALSE;
}
```

This code block is executed when a user clicks on any of the buttons in the page. It links the delete, refresh, save stream, and list box selection change events to their corresponding functions: `DeleteStream`, `RefreshStreams`, `SaveStream`, `SelectionChanged`. Each of these functions will be explained below.

RefreshStreams

The RefreshStreams function is the primary function of the property page. It is called when the page is first created to populate the list box with the names of the alternate data streams. It also populates the first line of the page which shows the total size of all of the alternate streams in the file. The source code for Refresh Streams is:

```
void RefreshStreams(HWND hwnd)
{
  static class ListFileContext {
    private:
      HINSTANCE hDll;
    public:
      NtQueryInformationFileFunc *pQIF;
      ListFileContext() {
        hDll = LoadLibrary ("NTDLL.DLL");
        pQIF = (NtQueryInformationFileFunc *)
        GetProcAddress (hDll, "NtQueryInformationFile");
      }
```

```
    ~ListFileContext() {
      FreeLibrary (hDll);
    }
  } Context;
```

As in the column handler, this code grabs the location of the NTQueryInformationFile function from NTDLL.DLL and stores it internally for later use.

```
HWND hwndList = GetDlgItem(hwnd, IDC_LIST);
SendMessageW(hwndList, LB_RESETCONTENT, 0, 0);
TCHAR wszStreamCount[100];
```

This code pulls the handle for the list box into the hwndList variable for later use and then clears the list box so new entries can be added. The last statement clears the variable that holds the text of the first line of the page, which normally displays the count and total size of the alternate streams

```
IO_STATUS_BLOCK Iosb;
unsigned char Buffer[2048];
FILE_INFORMATION_CLASS eClass;
FILE_STREAM_INFORMATION *pFsi;

HANDLE hFile = CreateFile( g_szFile, 0,
  FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
  OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, 0 );
```

These code blocks initialize some variables that will be used later and open a file handle to the file or directory. It is important to use the value “0” for the DesiredAccess argument to CreateFile so that the stream names can be viewed regardless of file lock status. Also, the flag `FILE_FLAG_BACKUP_SEMANTICS` must be set to allow CreateFile to return the handle of a directory.

```
if (hFile == INVALID_HANDLE_VALUE) {
  LoadString(_Module.GetModuleInstance(),
    IDS_FILELOCKED, wszStreamCount, 100);
  SetDlgItemText(hwnd, IDC_STREAMCOUNT, wszStreamCount);
  SendMessageW(hwndList, LB_SETCURSEL, 0, 0);
  SelectionChanged(hwnd);
  return;
}
```

This code checks to see if the file handle that has been returned is valid. If it is not, the resource IDS_FILELOCKED is loaded into the wszStreamCount variable that is then written to the first line of the page. In this case, IDS_FILELOCKED reads, “This object is unreadable, probably due to insufficient access.” If the file handle is invalid, execution is terminated at this point.

```
eClass = FileStreamInformation;
```

```
NTSTATUS rc = (Context.pQIF) (hFile, &Iosb,
  (PVOID) Buffer, sizeof(Buffer), eClass);

if (! Iosb.Information ) {
  LoadString(_Module.GetModuleInstance(),
    IDS_NOSTREAM, wszStreamCount, 100);
  SetDlgItemText(hwnd, IDC_STREAMCOUNT, wszStreamCount);
  SendMessageW(hwndList, LB_SETCURSEL, 0, 0);
  SelectionChanged(hwnd);
  return;
}
```

The first code block executes the NTQueryInformationFile function on the file handle that has been opened, returning the results into the stream information structure in `Buffer`. The second block tests to see that the `Iosb.Information` variable exists. If it does not, the handle is to a directory with no streams. In this case, the code writes the value of the resource `IDS_NOSTREAM` to the `wszStreamCount` variable that is written to the first line of the page. The IDS_NOSTREAM resource reads, "This object contains no alternate streams." If the handle is to a directory without a stream, the execution is terminated after the first line is written.

```
WCHAR wszStreamName[MAX_PATH];
LARGE_INTEGER totalStreamSize;
totalStreamSize.QuadPart = 0;
int streamCount = 0;
for (pFsi = (FILE_STREAM_INFORMATION *) Buffer;
  1;
  pFsi = (FILE_STREAM_INFORMATION *)
  ((BYTE *) pFsi + pFsi->NextEntryOffset)) {
```

Here the variables are initialized to store the name and size of a stream and the running count of alternate data streams found so far. The for loop will iterate through every stream in the file or directory, loading the stream information into the `pFsi` variable.

```
  ZeroMemory(wszStreamName, MAX_PATH);
  memcpy( wszStreamName,
    pFsi->StreamName,
    pFsi->StreamNameLength );
  wszStreamName[ pFsi->StreamNameLength/2 ] = 0;
```

The stream name variable space (`wszStreamName`) is cleared and then the name of the stream is copied out of `pFsi->StreamName`.

```
  if( wcsicmp( wszStreamName, L"::$DATA" )) {
    totalStreamSize.QuadPart =
      totalStreamSize.QuadPart + pFsi->StreamSize.QuadPart;
```

If the stream is not an unnamed stream ("::$DATA"), then the code should display it on the list box and include it in the total alternate stream size. The first statement after the if adds the stream size to the sum.

```
      LPWSTR pwsz = new WCHAR[MAX_PATH];
      lstrcpyW(pwsz, wszStreamName + sizeof(CHAR));
      LPWSTR wp = wcsstr(pwsz, L":");
      pwsz[wp-pwsz] = 0;
      lstrcpyW(wszStreamName, pwsz);
      delete [] pwsz;
```

The actual name of a stream is ":<stream name>:$DATA". The list box should only show the stream name and not the additional data, so this code trims out the stream name and stores it in `wszStreamName`.

```
      streamCount++;
      SendMessageW(hwndList,
       LB_ADDSTRING, 0, (LPARAM)wszStreamName);
    }
    if (! pFsi->NextEntryOffset)
      break;
  }
  CloseHandle( hFile );
```

This code block increments the counter of alternate data streams in the file or directory and then writes the name of the string into the list box. It then jumps back to the top of the for loop for the next stream and continues until all the streams are read. After all of the stream information has been copied to the list box, the file handle is closed.

```
char plurality[50] = {0};
char truncbytes[100] = {0};
char bytes[100] = {0};

if (streamCount == 1 ) {
  sprintf(plurality, "1 alternate stream");
} else {
  sprintf(plurality, "%lu alternate streams", streamCount);
}
```

When the code prints the first line of the page, it is important that the grammar is correct, so the code above selects the correct plurality of the word "stream" depending on whether there is one or more than one. It then stores either "1 alternate stream" or "<number> alternate streams" in the variable `plurality`.

```
if (totalStreamSize.QuadPart < 1024) {
  sprintf(truncbytes, "%s bytes:",
    Commify((double)totalStreamSize.QuadPart));
} else {
  sprintf(bytes, " (%s bytes):",
```

```
      Commify((double)totalStreamSize.QuadPart));
    if (totalStreamSize.QuadPart < 1048576) {
      sprintf(truncbytes, "%.2f KB",
        (double)totalStreamSize.QuadPart / 1024);
    } else if (totalStreamSize.QuadPart < 1073741824) {
      sprintf(truncbytes, "%.2f MB",
        (double)totalStreamSize.QuadPart / 1048576);
    } else {
      sprintf(truncbytes, "%.2f GB",
        (double)totalStreamSize.QuadPart / 1073741824);
    }
  }
```

The unattractive code block above insures that the total size of the streams is printed correctly. If the alternate stream is smaller than 1 kilobyte it should read "<number> bytes", between 1 kilobyte and one megabyte it should read "<number> KB", etc. The code also uses the Commify function like the column handler to ensure that the byte count is appropriately formatted.

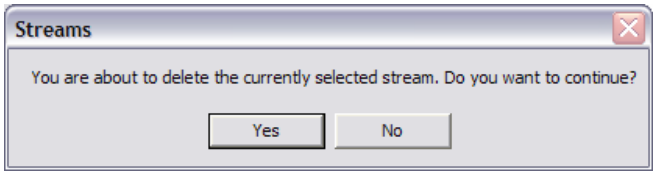
```
  if (streamCount > 0) {
    sprintf(wszStreamCount,
      "This object contains %s using %s%s",
      plurality, truncbytes, bytes);
  } else {
    LoadString(_Module.GetModuleInstance(),
      IDS_NOSTREAM, wszStreamCount, 100);
  }

  SetDlgItemText(hwnd, IDC_STREAMCOUNT, wszStreamCount);
  SendMessageW(hwndList, LB_SETCURSEL, 0, 0);
  SelectionChanged(hwnd);
}
```

Here the code finally prints either the string that has been assembled so far ("This object contains n alternate stream(s) using m bytes") or the IDS_NOSTREAM resource, depending on whether an alternate stream has been found in the file. The last statement calls the SelectionChanged function to update the line of text under the list box that displays the size of the currently selected stream.

DeleteStream

The DeleteStream function is called when the user selects a stream from the list box and then clicks the "Delete Stream" button. The function prompts the user with the confirmation dialog box in Figure 12 and then deletes the stream if appropriate.

Figure 12: The delete confirmation dialog box



The source code for DeleteStream follows:

```
void DeleteStream(HWND hwnd)
{
  TCHAR szStreamName[MAX_PATH];
  HWND hwndList = GetDlgItem(hwnd, IDC_LIST);
  int nCurSel = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
  SendMessage(hwndList, LB_GETTEXT, nCurSel, (LPARAM)szStreamName);
```

This code block finds out which alternate data stream the user has selected and stores the stream name in the variable `szStreamName`.

```
  // Prepare the fully-qualified stream name
  TCHAR szFileName[MAX_PATH*2];
  wsprintf(szFileName, _T("%s:%s"), g_szFile, szStreamName);
```

The filename `szFileName` for the stream is created from the combination of the path to the file and the name of the stream, separated by a colon.

```
  TCHAR szBuf[1024];
  LoadString(_Module.GetModuleInstance(), IDS_DELETE, szBuf, 1024);

  if (MessageBox(hwnd, szBuf, _T("Streams"), MB_YESNO)==IDYES) {
    DeleteFile(szFileName);
    RefreshStreams(hwnd);
  }
  return;
}
```

The string from the `IDS_DELETE` resource ("You are about to delete the currently selected stream. Do you want to continue?") is loaded into `szBuf` and displayed in a dialog box. If the user clicks Yes, the code will execute the DeleteFile function with the stream filename as the argument, deleting the stream without removing the file itself. The code then calls the RefreshStreams function to update the display.

SaveStream

SaveStream is called when the user clicks on the Save Stream button. It allows an alternate data stream to be saved to another file for analysis by programs that do not understand alternate data streams. This capability is extremely useful for forensic analysts that may want to save the stream to a non-

NTFS device or scan it with a virus or Trojan scanner. The SaveStream function pops up a standard Windows Save As box so the user can select the filename that they wish to save the stream to. The source code for SaveStream follows:

```
void SaveStream(HWND hwnd)
{
  TCHAR szStreamName[MAX_PATH];
  HWND hwndList = GetDlgItem(hwnd, IDC_LIST);
  int nCurSel = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
  SendMessage(hwndList, LB_GETTEXT, nCurSel, (LPARAM)szStreamName);
```

This code block finds out which alternate data stream the user has selected and stores the stream name in the variable `szStreamName`.

```
  // Prepare the fully-qualified stream name
  TCHAR szFileName[MAX_PATH*2];
  wsprintf(szFileName, _T("%s:%s"), g_szFile, szStreamName);
```

Just like in the DeleteStream function, the filename `szFileName` for the stream is created from the combination of the path to the file and the name of the stream, separated by a colon.

```
  OPENFILENAME ofn={0};
  TCHAR filename[MAX_PATH];
  filename[0]= 0;

  ofn.lStructSize     = sizeof(OPENFILENAME);
  ofn.lpstrFile       = filename;
  ofn.lpstrFilter     = "All Files (*.*)\0*.*\0\0";
  ofn.nMaxFile        = MAX_PATH;
  ofn.lpstrTitle      = TEXT("Save Stream As...\0");
  ofn.Flags           = OFN_READONLY |
                        OFN_PATHMUSTEXIST |
                        OFN_NOTESTFILECREATE |
                        OFN_OVERWRITEPROMPT;
```

This code sets up the parameters of the Save As box that is displayed to let the user choose the filename to save the alternate stream in. Some of the properties set above include the filter (in this case, All Files), the title of the save dialog, and a few flags to ensure that the save dialog performs as expected.

```
  if(GetSaveFileName((LPOPENFILENAME)&ofn))
    {
    DWORD nread, nwrite;
    TCHAR szBuf[1024];
```

If the user enters a valid filename into the dialog box, the GetSaveFileName function will store the name of the file selected in the `filename` variable. This code block also initializes some variables that will be used below.

```
    HANDLE infile = CreateFile(szFileName, GENERIC_READ,
      FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
      FILE_FLAG_BACKUP_SEMANTICS, NULL);
    if (infile == INVALID_HANDLE_VALUE)
      return;

    HANDLE outfile = CreateFile(filename, GENERIC_WRITE, 0,
      NULL, CREATE_ALWAYS, NULL, NULL);
    if (outfile == INVALID_HANDLE_VALUE)
      return;
```

Two file handles are opened, one for reading and one for writing. The filename for the reading handle is `szFileName`, which was assembled above, and the filename for the writing handle is `filename`, which was returned by the Save As dialog box.

```
    while (ReadFile(infile, szBuf, sizeof(szBuf), &nread, NULL) &&
      nread > 0) {
      WriteFile(outfile, szBuf, nread, &nwrite, NULL);
      }

    CloseHandle(infile);
    CloseHandle(outfile);
  }
  return;
}
```

The last bit of code here reads the data from the input handle and writes to the output handle, effectively copying the alternate data stream to the specified file. This method must be used because the standard copy functions do not understand the alternate stream syntax. The last statements make sure the file handles have been closed and then terminate execution.

SelectionChanged

The SelectionChanged function is called when the user selects a stream from the list box on the property page. The purpose of this function is to update the text below the list box to reflect the size of the selected stream. The source code is presented below:

```
void SelectionChanged(HWND hwnd)
{
  TCHAR szStreamName[MAX_PATH];
  HWND hwndList = GetDlgItem(hwnd, IDC_LIST);
  int nCurSel = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
  SendMessage(hwndList, LB_GETTEXT, nCurSel, (LPARAM)szStreamName);

  // Prepare the fully-qualified stream name
  TCHAR szFileName[MAX_PATH*2];
  wsprintf(szFileName, _T("%s:%s"), g_szFile, szStreamName);
```

```
  // Read the stream
  HANDLE hfile = CreateFile(szFileName, GENERIC_READ,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
    FILE_FLAG_BACKUP_SEMANTICS, NULL);
  if (hfile == INVALID_HANDLE_VALUE) {
    SetDlgItemText(hwnd, IDC_STREAMSIZE, "");
    return;
  }
```

As in the previous two functions, the code above opens a file handle to the stream that is selected by the user. If the file handle does not exist, the function immediately places an empty string into the area below the list box and terminates.

```
  DWORD fileSize;
  TCHAR fileSizePrint[200] = {0};
  CHAR fileSizeChange[100] = {0};
  fileSize = GetFileSize (hfile, NULL);
```

This code initializes a set of variables and uses the GetFileSize Windows API call to retrieve the size of the stream referenced by the handle. This size is returned into the variable `fileSize`.

```
  sprintf(fileSizePrint, "The size of the selected stream is ");
  if (fileSize < 1024) {
    sprintf(fileSizeChange, "%s bytes", Commify(fileSize));
  } else if (fileSize < 1048576) {
    sprintf(fileSizeChange, "%.2f KB (%s bytes)",
      (double)fileSize / 1024, Commify(fileSize));
  } else if (fileSize < 1073741824) {
    sprintf(fileSizeChange, "%.2f MB (%s bytes)",
      (double)fileSize / 1048576,   Commify(fileSize));
  } else {
    sprintf(fileSizeChange, "%.2f GB (%s bytes)",
      (double)fileSize / 1073741824, Commify(fileSize));
  }
  strcat (fileSizePrint, fileSizeChange);
  CloseHandle(hfile);
```

This code works like the code from RefreshStreams, properly formatting the size of the stream and writing it into the variable `fileSizePrint`, which is then concatenated with the string “The size of the selected stream is”.

```
  // display the content
  SetDlgItemText(hwnd, IDC_STREAMSIZE, fileSizePrint);
  return;
}
```

Finally, the function displays the line of text below the list box, which changes every time this function is called with a new argument.

Usage

Upon installation, an additional “Streams” tab will be added to the property page lists of every directory or file. If there is an alternate stream in the file or directory, it will appear in the list as illustrated in Figure 10. If there are no alternate streams in the file, the tab will still appear but the list box will be blank and the top text will read “This object contains no alternate streams.” The Refresh button still refreshes the list of streams in case one is added to the file while the property page is still open.

Conclusion

Alternate data streams play an important role in NTFS and should be utilized as a tool for more efficient data structuring. With new functionality comes new ways to use that functionality maliciously, but alternate data streams are not inherently dangerous. Using the shell extensions from this paper, a user or system administrator can integrate the usage of alternate data streams with everyday file browsing. Hopefully, this will bring alternate data streams out of the shadows and into the light.

Acknowledgements

I would like to thank Dino Esposito and Thomas Vogler for providing me with their source code to help put together the column handler and property page extensions. Dino’s article on NTFS provided a great resource with examples of implementing a stream-aware property page, and Thomas posted some excellent code to the NTDEV mailing list that showed me how to use the NTQueryInformationFile function correctly.

Bibliography

Esposito, Dino. “A Programmer’s Perspective on NTFS 2000 Part 1: Stream and Hard Link.” MSDN Library. March 2000.
URL: http://msdn.microsoft.com/library/en-us/dnfiles/html/ntfs5.asp

Kaspersky, Eugene and Zenkin, Davis. “NTFS Alternate Data Streams.” Storage Admin, Windows and .NET Magazine. Spring 2001.
URL: http://www.storageadmin.com/Articles/Index.cfm?ArticleID=19878

Kozierok, Charles M. “Master File Table (MFT).” PC Guide. 17 April, 2001.
URL: http://www.pcguide.com/ref/hdd/file/ntfs/archMFT-c.html

LeBlanc, David. “Detecting Alternate Data Streams.” Security Administrator, Windows and .NET Magazine. 30 November, 2000.
URL: http://www.secadministrator.com/Articles/Index.cfm?ArticleID=16189

MSDN Library. URL: http://msdn.microsoft.com

“NTFS Documentation.” 11 February, 1999.
URL: http://www.scit.wlv.ac.uk/~cm1924/cp3025/filesys/reading/ntfs6/index.html

Russinovich, Mark. “Inside Win2K NTFS, Part 2.” Windows and .NET Magazine. Winter 2000.
URL: http://www.win2000mag.net/Articles/Index.cfm?ArticleID=15900

Russinovich, Mark. “Inside NTFS.” Windows and .NET Magazine. January 1998.
URL: http://www.winntmag.com/Articles/Index.cfm?ArticleID=3455

“Size Limitations in NTFS and FAT File Systems.” Microsoft TechNet.
URL:
http://www.microsoft.com/technet/prodtechnol/winxppro/reskit/prkc_fil_tdrn.asp

“Virus Scanner Inadequacies with NTFS.” Dartmouth’s Institute for Security Technology Studies. 18 August, 2000.
URL: http://www.ists.dartmouth.edu/IRIA/knowledge_base/NTFS_Advisory.htm

Vogler, Thomas. “somewhat working sample to enumerate ntfs file streams.” NTDEV Mailing List. 15 January, 1998.
URL: http://www.ntdev.org/archive/ntdev9801/msg0216.html