A Win32-Based Technique for Finding and Hashing NTFS Alternate Data Streams

Alden Dima
National Software Reference Library
National Institute of Standards and Technology
alden.dima@nist.gov

Abstract

As part of an effort to create new datasets for the computer forensic community, the National Software Reference Library created a simple Windows specific tool for internal use similar to md5deep, which is able to recursively walk a New Technology File System (NTFS) and generate cryptographic hashes for each file encountered. Unlike md5deep, our tool is able to find and hash any alternate data streams associated with file and directories.

Many computer security and forensics professionals view NTFS alternate file streams as a threat because they allow evidence to be hidden within the file system “behind” ordinary files and directories. At the same time, many common applications and utilities use alternate data streams to hold metadata. We needed the tool to find and hash all of the alternate data streams associated with each file and directory in the file system. This will help us identify alternate data streams that are benign and can be safely ignored by forensic investigators. We also desired a solution that relies only on the standard Win32 application programming interface (API) and avoids using the nonstandard native Windows API that is subject to change with each Windows release.

This paper will discuss the key implementation issues of this tool and shows that the key issue with writing tools to process alternate data streams lies in initially finding the alternate data streams and not with their subsequent processing. While this information has been previously available inside the Windows system programming community, we hope to expose the forensic tool developers to the key principles of writing tools that work with alternate data streams and thus encourage developers to add this functionality to their projects.

Introduction

Modern personal computer file systems can contain a large amount of information; a typical PC usually contains between 10,000 and 100,000 files [1]. The huge amount of data involved creates a burden for forensic investigators as they sift through files looking for evidence. Fortunately the use of software that can generate cryptographic hashes for each file in a file system coupled with databases of cryptographic hashes can greatly reduce the forensic investigator's workload [1]. Once such tool is md5deep. It is a full-featured tool that can recursively process entire directory structures in a live file system and generate cryptographic hashes for each file [2].

While md5deep is portable across a wide-variety of platforms, it and many other similar portable tools that work with live file systems have one weakness: they do not process Microsoft Windows New Technology File System (NTFS) alternate data streams. This can be readily verified by creating alternate data streams associated with files in a directory and having md5deep or similar tools hash the contents of the directory. This weakness is not really an intrinsic flaw of these tools, but more a function of the unique nature of NTFS alternate data streams and the constraints placed on developers of the portable system libraries used by these tools.

The New Technology File System and Alternate Data Streams

The New Technology File System is the native file system format for the Windows NT family of operating systems (Windows NT, 2000, XP, and Server 2003) [3]. It implements a number of advanced features not found in the File Allocation Table (FAT) file systems used in earlier version of Windows. These features include improved file retrieval from large directories as well as improved security, logging, recoverability, and disk utilization [4].

To flexibly implement its advanced features, NTFS treats files as a collection of attribute/value pairs; each file property is represented as an NTFS object attribute that consists of a single stream or sequence of bytes. This open-ended approach makes it simple to add additional attributes to a file. A file's data is just another attribute of the file and is implemented in no differently than the other attributes. The default data attribute has no name and serves as what most users consider to be the file's data. Because attributes are easy to add, a file can contain multiple data attributes. These extra data attributes exist as alternate data streams (ADS) and are given names during creation to distinguish them from default data attribute and each other. [3]

From a programmer's perspective, each stream is a separate file. It has for example, its own allocation size and file lock. To simplify the Win32 application programming interface, the full name of an alternate data stream is specified by appending a colon to the file name followed by the stream name. This means that many of the Win32 functions treat ADS as just another file. [3][5]

Unfortunately, for Windows XP and earlier, ADS support at the Win32 level is not complete. While known alternate data streams can be handled as ordinary files, there is no easy way to list all of the alternate data streams for a given file. Although the most recent version of Windows, Windows Server 2003 does offer two functions *FindFirstStreamW* and *FindNextStreamW* which can enumerate all of the streams for a given file or directory, these Win32 functions are not available for Windows XP. As a result, for the vast majority of Windows-based PCs, a different approach is necessary. [6][7]

Examples of Legitimate ADS Use

There are a number of legitimate uses of NTFS alternate data streams; many of them are directly related to the Windows internal operations. Since their introduction in 1993 with Windows NT 3.1, the subsequent Windows releases have increasingly relied on their presence. [8]

The original application for alternate data streams was to allow Apple Macintosh clients to use Windows NT-based file servers [8]. The file server support in Windows Server for Classic Apple Macintosh operating systems uses alternate data streams to store the resource fork associated with Apple files. [3][8]

With the release of Windows 2000, the Windows Explorer started saving user-supplied file summary data entered via the Properties tag as an alternate data stream attached to the file. [3][8]

Microsoft also implemented NTFS volume change tracking in Windows 2000. This feature uses alternate data streams to store volume changes in a change journal and is particularly useful for system utilities like anti-virus programs and backup tools. [8]

Windows XP Service Pack 2 introduced Attachment Execution Services (AES) that gives each email attachment a risk rating. When the attachment is saved to disk, its risk rating metadata is saved as an alternate data stream. [9][10]

Examples of ADS-Based Threats

Despite their legitimate applications, alternate data streams are an excellent means for hiding data covertly. Data that can be normally stored in a file in the typical fashion (via the default data stream) can also be stored in an alternate data stream. As a result, ADS can provide a covert channel for criminal activity [8]. Given that there are legitimate applications that use alternate data streams with known names, a malicious user can create alternate data streams that mimic legitimate ADS to further disguise criminal activity [8].

Executable code can also be stored in an alternate data stream [8]. Malware has been hidden in alternate data streams [11]. Alternate data streams provide computer criminals with a means of hiding root kits or system cracking tools on a compromised system and facilitate their undetected execution [12].

Executable code running from an ADS will appear in the Windows Task Manager under the name of the file to which the ADS is associated. This makes it easy to disguise the execution of an illegitimate process by attaching the its executable code as an ADS associated with a normally legitimate file. [12]

ADS Detection Issues

The key ADS-related issue from a computer forensic/security point-of-view is their relative invisibility to the system tools provided with Windows [8]. The standard Windows command-line utilities, the standard Windows File management tools, and Windows backup programs offer poor support for alternate data streams. [12][13]. There are some system tools that can create and manipulate alternate data streams, but there are no standard Windows tools that can find unknown alternate data streams [8].

There is a small number of readily available command line tools for finding ADS and these tools are often limited in functionality [13]. The ability to detect the presence of ADS is not enough; a forensic analyst needs to be able to examine the contents of each stream [8]. This implies the need for tools that can process the stream contents to produce cryptographic hashes or perform various searches. As noted earlier, while there are tools such as md5deep that can recursively hash all of the files in a file system, many cannot process alternate data streams.

Professional computer forensic tools can handle alternate data streams, but their mode of operation is often slow and less efficient when compared to the tools available on a “live” system [13]. Currently, forensic investigations typically follow the pattern of on-site capture of system data followed by laboratory analysis of the captured data. While this provides for a thorough means of analyzing, handling and preserving evidence, there are situations where less resource intensive means are desired. [14]

Installation Hashing

Our interest in hashing alternate data streams started when we began to investigate installation hashing as part of our emerging Windows Registry Dataset effort (WIRED). WIRED will be a dataset of Windows Registry changes caused by the installation, removal and operation of software of forensic interest. Because we will be installing software as part of the work, we decided to also capture hashes of the files installed or modified during the software installation process. Given the increasing use of alternate data streams by Windows applications, we decided to find and hash any alternate data streams associated with the installation of the software.

When we did not find tools that met our specific needs, we investigated writing our own. We sought a solution that relied only on the standard Win32 API and avoided using the nonstandard native Windows API that is subject to change with each Windows release. It turned out to be a relatively straightforward task. We then decided to encourage other forensic tool builders to add alternate data stream awareness to their tools.

The Basic Algorithm

Our code for iteratively walking an NT file system and hashing each data stream consists of three parts; two of which are fairly generic. These parts are directory traversal, stream enumeration, and stream hashing. Only the enumeration of the streams on Windows XP or earlier can be considered somewhat tricky.

Directory Traveral

The top-level code concerns itself with finding and traversing all of the directories found starting from either the current directory or some user-specified directories. Its structure in pseudocode as follows:

```
TOP LEVEL:

Set Process Privileges for Backup

IF no directory specified THEN
   Get current directory
   Save it for later traversal
   Enumerate and hash its streams
ELSE
   FOR EACH directory specified LOOP
      Put the directory path in canonical form
      Save it for later traversal
      Enumerate and hash its streams
   END LOOP
END IF

FOR EACH directory saved for traversal LOOP
   Clean up path value if needed

   FOR EACH item in the directory LOOP
      IF it's a directory but not a dot directory THEN
         Save it for later traversal
      END IF
      Enumerate and hash its streams
   END LOOP
END LOOP
```

Even though the overall structure is fairly simple, there are three potential problem areas. The first is that Windows built-in security can prevent user-mode programs such as backup utilities or file system walkers from accessing many of the files on the system. Fortunately Windows provides a mechanism that allows certain programs run by the Administrator to essentially ignore the standard security checking. This feature is enabled at runtime by obtaining the process access token and adding the *SE_BACKUP_NAME* privilege to it. When each file is subsequently opened for reading using *CreateFile* and with the *FILE_FLAG_BACKUP_SEMANTICS* flag in the *dwFlagsAndAttributes* parameter, the process will be treated as a backup process and be allowed to override the standard security checking. [15]

The process access token is obtained via a call to *OpenProcessToken* [16].
The Microsoft Developer Network Library provides a C++ function, *SetPrivilege*, as example code [17]. This function can be called with the process access token, the name of the privilege and a privilege status to either enable or disable that privilege for the process. The code to handle these security-related issues is structured as follows [18]:

```
HANDLE hToken = NULL;

if (!OpenProcessToken(GetCurrentProcess(),
TOKEN_ADJUST_PRIVILEGES, &hToken))  {
    /* handle error */
}

if (!SetPrivilege(hToken, SE_BACKUP_NAME, TRUE)) {
    /* handle error */
}

if (hToken) {
    CloseHandle(hToken);
}
```

The second issue is that it is also easy to forget that an NTFS directory is little more than a file whose default data stream contains directory information. At least one commercial ADS detection tool does not detect alternate data streams attached to a directory [8]. Alternate data streams can be attached to a directory just like they can be attached to an ordinary file. As a result, each directory must be processed as though it were a file while looking for alternate data streams.

The third issue is that directory paths entered at the command line need to be verified and put in a consistent format. The traversing and hashing of directory trees can take some time; an inaccessible directory at the command line should be detected as early as possible. Putting the directories entered at the command line into a consistent format simplifies later comparison of the results. The Windows command line shell is case insensitive yet case-preserving and also does not fully expand the paths it passes as command-line parameters to programs it executes. The following code shows the method chosen to deal with these issues. This greatly simplifies file system comparisons made between different executions of our tool:

```
BOOL GetCurrentDirectoryX(path_t &curr_dir) {
    DWORD dwRet = GetCurrentDirectoryW(MAX_PATH + 1,
        curr_dir.value);
    return (!dwRet || dwRet > MAX_PATH + 1) ?
        FALSE : TRUE;
}
```

```
BOOL CanonifyDirectoryPath (
    const path_t &curr_dir,
    const path_t &in_path,
    path_t &cannon_path)
{
    return SetCurrentDirectory(in_path.value)
        && GetCurrentDirectoryX(cannon_path)
        && SetCurrentDirectory(curr_dir.value);
}
```

The basic idea is to change the current directory to the directory in question and then ask Windows for the current directory. Windows will then return the directory path as an absolute path in a consistent fashion.

Enumerating Streams

Despite the lack of complete support of alternate data streams afforded by Win32 as implemented in Windows XP and earlier, there is a slightly quirky way to use existing Win32 functions to enumerate all of the streams associated with either a file or directory [4][8]. The following is an outline of the method used:

```
ENUMERATE STREAMS:

IF the path is for a file THEN
   Save the path as the first stream
END IF

Open object represented by the path for reading

LOOP
   Using BackupRead, read the first part of the stream
   header

   IF there is nothing to read THEN
      Stop looping
   END IF

   Determine whether the header belongs to an ADS
   Determine the size of its data
   Get the trailing part of the stream header

   IF it is an alternate data stream THEN
      Get the stream's name from the trailing part
      Save the full path with the stream name for hashing
   END IF

   Using BackupSeek with the stream's data size,
   find the next stream header
END LOOP

Use BackupRead with special parameters to stop reading
the object

Close the object
```

This method relies on the *BackupRead* and *BackupSeek* functions provided in Win32 as a means of writing backup utilities. In order to allow users to properly backup all data associated with an NTFS file, *BackupRead* will read the file as a single stream of bytes in which all of the file's streams are separated by headers that describe the data that follows them [19]. The stream headers are formatted as C structures of type *WIN32_STREAM_ID* that have the following definition [20]:

```
typedef struct _WIN32_STREAM_ID {
  DWORD dwStreamId;
  DWORD dwStreamAttributes;
  LARGE_INTEGER Size;
  DWORD dwStreamNameSize;
  WCHAR cStreamName[ANYSIZE_ARRAY];
} WIN32_STREAM_ID,
 *LPWIN32_STREAM_ID;
```

The intention of this structure is to have the fixed size header data followed by a variable sized stream name. Note that the constant *ANYSIZE_ARRAY* is defined to be 1. This value was chosen as the smallest legal array size before zero-length arrays became a legal standard C construct in 1999. Windows predates this language change hence the value of *ANYSIZE_ARRAY*. [21]

To facilitate the handling of the stream header, a header structure can defined as follows:

```
typedef struct stream_header {
    WIN32_STREAM_ID id;
    WCHAR name_tail[MAX_PATH + 1];
} stream_header_t;
```

This allows us to easily deal with the trailing stream name without having to juggle bytes.

The alternate data stream enumeration proceeds as follows. The file or directory to be examined for ADS is opened using *CreateFile*:

```
hIn = CreateFileW(file_path.value, GENERIC_READ, FILE_SHARE_READ,
NULL, OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, NULL);
```

Note that *CreateFileW* is the wide-character variant of *CreateFile*. The wide character Win32 functions were used because NTFS stores file names in Unicode format and we didn't want to risk missing files due to character mapping issues. Many of these issues can be dealt with transparently by using Microsoft-provided macros [22], but we opted to take a more explicit route.

Once opened the fixed-size portion of the first stream header is read:

```
BackupRead(hIn, (LPBYTE) &stream_header, SH_FIRST_SZ,
&dwReadCount, FALSE, TRUE, &pReadContext)
```

Three important stream properties can now be determined from the header. The first is the stream ID. For alternate data streams, the *dwStreamId* field will have the *DWORD* value 0x00000004 (*BACKUP_ALTERNATE_DATA*) [20]. The second is the stream's data size in bytes. The Size field will contain its value as a *LARGE_INTEGER* value. The third is the stream name size, contained in *dwStreamNameSize* as a *DWORD* value.

The stream name must still be read from the variable sized portion of the stream header. This is accomplished this by moving forward in the stream header buffer by the size of the fixed portion of the stream header while using *BackupRead* to read exactly *dwStreamName* bytes from the opened file:

```
BackupRead(hIn, (LPBYTE) &stream_header + SH_FIRST_SZ,
dwStreamNameSize, &dwReadCount, FALSE, TRUE, &pReadContext)
```

For each alternate data stream found, the full absolute path to the stream is saved in the *drive:\path\filename:stream* format [5]. These paths can later be processed just like regular files.

Because we are only interested in the default and alternate data streams, and because the default stream can be accessed without *BackupRead*, we'll ignore all of the non-ADS streams found by *BackupRead* and use *BackupSeek* to skip ahead:

```
BackupSeek(hIn, cbStreamSize.LowPart, cbStreamSize.HighPart,
&dwLow, &dwHigh, &pReadContext);
```

Note the stream offsets used by *BackupSeek* are really 64-bit quantities and are implemented as a pair of *DWORD* parameters. The parameters *dwLow* and *dwHigh* are used as return values to give the actual offset achieved by *BackupSeek*. [23]

Hashing Alternate Data Streams

Once the alternate data streams have been found, they can be processed just like an ordinary file. For example, if a file *test.txt* has two alternate data streams *test.txt:stream1* and *test.txt:stream2*, then for our purposes there are three files that must be read using *ReadFile*. The only difference in handling the alternate data streams versus a regular file will be that the path to the alternate data stream passed as a parameter to *CreateFile* has the stream name appended to it. From that point on, the mechanics are exactly the same.

Sample Output

Because alternate data streams are easy to create, it is a simple task to create a test directory containing files with alternate data streams. For example, a test directory was created containing both files and directories that have associated alternate data streams. To the *DIR* command, this directory appears as follows:

```
 Directory of d:\ADS-test

10/18/2006  01:30 PM    <DIR>          .
10/18/2006  01:30 PM    <DIR>          ..
08/03/2006  04:39 PM                20 data.txt
08/07/2006  11:42 AM           164,040 hash.exe
08/04/2006  01:06 PM                 0 none.txt
07/27/2006  02:05 PM                 0 none2.txt
07/28/2006  02:54 PM    <DIR>          secret
07/27/2006  01:55 PM            36,864 streams.exe
               5 File(s)        200,924 bytes
               3 Dir(s)  48,726,573,056 bytes free
```

There are no hints of alternate data streams other than the suggestive names of certain files. According to the *DIR* command, the "secret" subdirectory contains the following:

```
 Directory of d:\ADS-test\secret

07/28/2006  02:54 PM    <DIR>          .
07/28/2006  02:54 PM    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  48,726,573,056 bytes free
```

The output from our program reveals otherwise:

```
16D626CBAF24ACF1EABA8309805BB1008A994A72   d:\ADS-test:secret
16D626CBAF24ACF1EABA8309805BB1008A994A72   d:\ADS-test\data.txt
AF49D1C0672C8307F5842C7358F3B6E26F772A87   d:\ADS-
test\data.txt:□SummaryInformation
DA39A3EE5E6B4B0D3255BFEF95601890AFD80709   d:\ADS-
test\data.txt:{4c8cc155-6c1e-11d1-8e41-00c04fb9386d}
D23FA31158082B1EA2D244A550C51E132F85D002   d:\ADS-test\hash.exe
D59FC84CDD5217C6CF74785703655F78DA6B582B   d:\ADS-
test\hash.exe:Zone.Identifier
DA39A3EE5E6B4B0D3255BFEF95601890AFD80709   d:\ADS-test\none.txt
1909263AB8B1753936FE2C4A4406299098FD2750   d:\ADS-
test\none.txt:□SummaryInformation
DA39A3EE5E6B4B0D3255BFEF95601890AFD80709   d:\ADS-
test\none.txt:{4c8cc155-6c1e-11d1-8e41-00c04fb9386d}
DA39A3EE5E6B4B0D3255BFEF95601890AFD80709   d:\ADS-test\none2.txt
16D626CBAF24ACF1EABA8309805BB1008A994A72   d:\ADS-
test\none2.txt:alden
DA39A3EE5E6B4B0D3255BFEF95601890AFD80709   d:\ADS-
test\none2.txt:none
16D626CBAF24ACF1EABA8309805BB1008A994A72   d:\ADS-
test\secret:secret
F8E85CDF9ED7539A130D555A9A26E2A9972EFD6C   d:\ADS-test\streams.exe
D59FC84CDD5217C6CF74785703655F78DA6B582B   d:\ADS-
test\streams.exe:Zone.Identifier
```

Note: The output is wrapped here because of space constraints. The first column is the SHA-1 hash value for the contents of the file or alternate data stream. The second column shows the absolute path to the file or alternate data stream. The presence of the stream SHA-1 values is extremely useful in comparing the contents of the stream with known files. These known files are typically available as collections of cryptographic hash values such as the NSRL Reference Dataset [1].

A number of interesting items appear in the output. For example, both the directory *ADS-test* and its subdirectory *secret* have alternate data streams associated with them. It is easy to forget that ADS can be associated with directories as well as files. Some of the files have alternate data streams named *SummaryInformation* (with a leading non-ASCII character). These ADS were created using the Windows Explorer with user-supplied file summary data entered via the Properties tag (see [3][8]). The alternate data streams named *Zone.Identifier* were created by the Windows XP Attachment Execution Services when the files were download from the Internet [10][24][25]. The streams named *secret* were created at the Windows command prompt using the method described in [12]. For example, the stream *d:\ADS-test:secret* was created using:

```
    type secret > d:\ADS-test:secret
```

where *secret* is a text file containing some arbitrary text. None of these alternate data streams are visible from command-line via the *DIR* command, nor do they appear in the Windows Explorer.

Future Windows Releases

The method outlined above to enumerate alternate data streams is but a temporary fix. Microsoft documentation reveals that post Windows XP releases (Windows Server 2003 and later) offer the ability to handle alternate data streams directly via the Win32 API [6][7][26]. This new functionality appears to be centered on the functions *FindFirstStreamW* and *FindNextStreamW* and the structure *WIN32_FIND_STREAM_DATA* [6][7][26].

Early users of Windows Vista have also reported that the *dir* command offers a command-line switch to display alternate data streams [27][28]. These developments suggest that in the future, alternate data streams will become a more visible part of day-to-day Windows usage and less of a secret, obscure operating system feature.

Conclusion

Despite their reputation as a means of hiding covert data, writing code to handle alternate data streams in Windows XP (and earlier) is rather straightforward. Though it is not their intended purpose, the *BackupRead* and *BackupSeek* functions provide the necessary means to find all of the alternate data streams associated with a file or directory. Once an alternate data stream is found, it can be treated just like a regular file.

Future versions of Windows promise to make the task of writing tools that handle alternate data streams even easier. We hope that the work presented here will encourage forensic tool developers to make their tools ADS-aware and as a result, effectively remove alternate data streams from the toolbox of computer criminals.

References

[1] "Project Overview," *National Software Reference Library*, September 2006; http://www.nsrl.nist.gov/Project_Overview.htm.

[2] "md5deep," September 2006; http://md5deep.sourceforge.net/.

[3] M.E. Russinovich and D.A. Solomon, "Microsoft Windows Internals," 4th ed., Microsoft Press, 2005.

[4] D. Esposito, "A Programmer's Perspective on NTFS 2000 Part 1: Stream and Hard Link," *Microsoft Developer Network Library*, March 2000; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnfiles/html/ntfs5.asp.

[5] "How To Use NTFS Alternate Data Streams," *Microsoft Knowledge Base*, July 2006; http://support.microsoft.com/kb/105763.

[6] "FindFirstStreamW," *Microsoft Developer Network Library*, August 2006; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/findfirststreamw.asp.

[7] "FindNextStreamW," *Microsoft Developer Network Library*, August 2006; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/findnextstreamw.asp.

[8] R.L. Means, "Alternate Data Streams: Out of the Shadows and into the Light," SANS Institute, 2003; http://www.sans.org/reading_room/whitepapers/honors/1503.php.

[9] R. Larno, "Windows XP SP2 for Developers", Microsoft Benelux, September 2004; http://download.microsoft.com/download/e/1/e/e1e86586-a01e-41e9-91a9-6a12ded994e4/04-08-31_Windows_XP_SP2_A_Developer_Overview.ppt.

[10] "Description of how the Attachment Manager works in Windows XP Service Pack 2," *Microsoft Knowledge Base*, August 2006; http://support.microsoft.com/?kbid=883260.

[11] S. Baker, P. Green, T. Meyer and G. Cochrane, "Checking Microsoft Windows 1Systems for Signs of Compromise", ver. 1.3.4, Univ. Col. London Computer Security Team, 28 October 2005; http://www.ucl.ac.uk/cert/win_intrusion.pdf.

[12] R. Zadjmool, “Hidden Threat: Alternate Data Streams,” *WindowSecurity.com*, 23 July 2004; http://www.windowsecurity.com/articles/Alternate_Data_Streams.html.

[13] D. Bem and E. Z. Huebner, “Alternate Data Streams in Forensic Investigations of File System Backups,” *2nd Intl. Conf. on Comp. Sci. & Info. Sys.*, Athens, Greece, 19-21 June 2006; http://www.cit.uws.edu.au/compsci/computerforensics/Technical%20Reports/ADS%20in%20Forensics%20Investigations%20of%20Backups%20Web%20version.pdf.

[14] C. Blankenhorn, E. Huebner and M. Cook, “Forensic Investigation of Data in Live High Volume Environments,” *UWS Col. of Sci. Tech. and Environment Innovation Conf. 2005*, Univ. of Western Sydney, Penrith, Australia, 2005.

[15] “File Security and Access Rights,” *Microsoft Developer Network Library*, October 2006; http://windowssdk.msdn.microsoft.com/en-us/library/ms685569(VS.80).aspx.

[16] “OpenProcessToken,” *Microsoft Developer Network Library*, October 2006; http://windowssdk.msdn.microsoft.com/en-gb/library/ms722901(VS.80).aspx.

[17] “Enabling and Disabling Privileges in C++,” *Microsoft Developer Network Library*, October 2006; http://windowssdk.msdn.microsoft.com/en-us/library/ms719308(VS.80).aspx.

[18] “Taking Object Ownership in C++,” *Microsoft Developer Network Library*, October 2006; http://windowssdk.msdn.microsoft.com/en-us/library/aa379620(vs.80).aspx.

[19] “BackupRead,” *Microsoft Developer Network Library*, July 2006; http://msdn.microsoft.com/library/en-us/backup/backup/backupread.asp.

[20] “WIN32_STREAM_ID,” *Microsoft Developer Network Library*, July 2006; http://msdn.microsoft.com/library/en-us/backup/backup/win32_stream_id_str.asp.

[21] “The Old New Thing: Why do some structures end with an array size of 1?,” *Microsoft Developer Network Blogs*, July 2006; http://blogs.msdn.com/oldnewthing/archive/2004/08/26/220873.aspx.

[22] J. M. Hart, “Windows System Programming,” 3rd ed., Addison-Wesley, 2005.

[23] “BackupSeek,” *Microsoft Developer Network Library*, July 2006; http://msdn.microsoft.com/library/en-us/backup/backup/backupseek.asp.

[24] “Persistent Zone Identifier Object,” *Microsoft Developer Network Library*, October 2006; http://msdn.microsoft.com/workshop/security/szone/reference/

objects/persistentzoneidentifier.asp.

[25] B. De Smet, “Demo of Attachment Execution Service Internals in Windows XP SP2 AND Windows Server 2003 SP1,” B# .NET Blog, October 2006; http://community.bartdesmet.net/blogs/bart/archive/2005/08/19/3485.aspx.

[26] “WIN32_FIND_STREAM_DATA,” *Microsoft Developer Network Library*, August 2006; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/win32_find_stream_data_str.asp.

[27] B. De Smet, “Use Vista's DIR command to display alternate data streams,” B# .NET Blog, August 2006; http://community.bartdesmet.net/blogs/bart/archive/2006/07/13/4129.aspx.

[28] J. Zhang, “Junfeng Zhang's .Net Framework Notes,” August 2006; http://blogs.msdn.com/junfeng/archive/2006/04/21/580285.aspx.