BrianHetrick.com
Application Note AN-10

A Temporary Stream Provider for .NET

Contents

Goal....1
Introduction....1
Implementation....3
Test Program....5
Summary....5
Further Information....6
Disclaimer....6
Copyright and Grant of License....7

Goal

This Application Note describes a temporary stream provider for .NET applications. It discusses the requirements upon for temporary streams, the safeguards that must be associated with temporary streams, and an implementation approach meeting these requirements. It also discusses needs for temporary storage that cannot be met by simple streams, and how provisions for these more demanding requirements can be made within a temporary stream facility. It presents an implementation for temporary streams directly usable in .NET applications. Finally it presents a sample application using the temporary stream facility.

The .ZIP file accompanying this Application Note includes both the facility and the sample application used to demonstrate the facility.

Introduction

A stream is typically defined as an ordered sequence of byte values, which can be read or written by a computer program. Typical access to a stream is sequential: bytes are read or written one after the other, with the position of the "current" byte in the stream increasing by one with each byte read or written. Some streams may be seekable, that is, the position of the "current" byte in the stream can be set with a seek operation. Streams might be durable or evanescent: an evanescent stream, such as a network or serial communications link, exists only when it is being both read and written, while a durable stream, such as a disk file, continues to exist after being written even it if has not been read.

Functionally, streams differ from, for example, List<> objects in that they have effectively infinite capacity. Evanescent streams are typically consumed as they are produced, and storage limits typically do not come into play at all. Evanescent streams are typically not seekable: at any given time, the manifestation of the stream is limited to bytes produced but not yet consumed; skipping ahead to bytes not yet produced, or rereading bytes already consumed, is typically impossible. Durable streams are typically realized on secondary storage, for which capacity limits are currently typically in the terabyte range rather than the gigabyte range of current typical primary storage. Durable streams can be, but need not be, seekable. Typically, in order to be efficiently seekable, a durable stream must be both uncompressed and stored on random access media.

One particular type of durable stream is the temporary stream. A temporary stream is used to provide storage for information both generated and consumed by a single execution of a single program. Temporary streams in this sense are not used for passing information from one program to another, or from one execution of a program to a later execution of the same program. The lifetime of a temporary stream is bound up with the lifetime of the program that created it: as a temporary stream is useful only for a single execution of a single program, it should be removed when the creating and using program exits. However, viewing the Windows conventional location for temporary files (in Windows 7 and later, typically `C:\Users\`*username*`\AppData\Local\Temp`) will frequently reveal hundreds or thousands of abandoned temporary files, many with revealing names. Relatively few applica-

tions are careful about cleaning up after themselves or withholding metadata identifying their temporary data.

Temporary streams are commonly used for temporary storage of arbitrarily large amounts of data. A temporary stream might be used for holding a pre- or post-processing copy of a file to be updated in place; or for holding XML documents involved in a network service request or reply; or for holding a log of messages produced during a lengthy process where the log is intended to be discarded upon successful completion of the process and reported only if the process exceptionally encounters errors. Temporary databases, relatives of temporary streams, might be used to hold indexed caches of data or complex intermediate processing state.

Temporary streams have security implications. Data security is typically considered to include confidentiality, integrity, and accessibility. A hostile program might read another program's temporary stream, attacking confidentiality. A hostile program might observe the size or meaningful name of another program's temporary stream, permitting correlation with known inputs and hence enabling traffic analysis. A hostile program might write to another program's temporary stream, attacking integrity. A hostile program might lock another program's temporary stream, attacking availability. If a privileged program uses temporary streams and predictable temporary stream file names, a hostile program might create a file system entry with the predictable name which later causes the privileged program to overwrite a system file with its temporary stream. A hostile program might force a program to exit prematurely, leaving its temporary streams available for extended analysis. Some of these security vulnerabilities can be mitigated to an extent. In the absence of a capability-based operating system, however, attacks from hostile programs running as the same user account as the attacked program are problematic.

Confidentiality of temporary streams can be guaranteed by encrypting them with a nonce that is not revealed. Integrity of temporary streams can be enhanced by compressing the plain text of the stream content: addenda and substitutions to the temporary stream will likely cause post-decryption decompression to fail or to produce uninterpretable nonsense. This also mitigates to an extent against stream size observation, although compression ratio attacks are still possible. Availability of temporary streams can be guaranteed by leaving them open with sharing prohibited throughout their lifetimes on Windows, or by creating them with access mask 0 and leaving them open throughout their lifetimes on Unix-like systems. Another technique available on Unix-like systems is to create a file and immediately unlink (delete) it while leaving the file open throughout its lifetime: this removes the name of the file, making it inaccessible to other programs, but delays deleting the file itself until it is closed. These availability defense techniques have both stream usage complications and resource consumption implications for the temporary stream owning program, which argues against them in most circumstances. Avoiding predictable names for streams' backing files mitigates against the system file overwrite attack.

Best practice in the use of temporary streams involves removing them when they are no longer in use, and avoiding meaningful names for backing files. Both are enhanced by creating a subdirectory in the user's temporary file directory, and placing any temporary files in that subdirectory. Both the subdirectory and any temporary files should be named with random, unpredictable names.

Compression techniques can be used only with sequential access patterns: the ability to interpret compressed data at a point in the compressed stream depends upon knowledge of the decompressed stream up to that point. Seeking to an arbitrary point and reading the data there is effectively impossible. Some types of files, such as databases, cannot be implemented with non-seekable files. However, facilities providing application-private data storage services (such as SQL Server Compact Edition, or SQLite with the SQLite Encryption Extension, or the System.Data.SQLite .NET interface to SQLite) typically offer both on-disk encryption with a user-specified key and user-specified file names for on-disk files. In this circumstance, a temporary stream facility can be expanded to a temporary file facility by providing access to the private key nonce generator and file name generator.

This Application Note describes a temporary sequential-only access stream facility for .NET programs with most of the security mitigations described above. Stream contents are compressed and encrypted; files containing stream contents are assigned meaningless unpredictable names in a meaningless and unpre-

dictably named subdirectory of the user's temporary directory; files containing stream contents are discarded when the controlling application object is disposed; and the nonce and file name generators are exposed to consumers. Due to the compression, the temporary streams are not seekable. The streams are not left open between creation and later consumption, permitting the attack against availability; permitting this vulnerability both decreases resource consumption and permits multiple readers of the temporary stream.

Implementation

The overall structure of the temporary stream provider is a static class jacket to a singleton instance of an implementation class. Only one implementation class is provided. This singleton implementation technique is described in The Singleton Pattern in .NET.

The work stream provider implements the `IWorkStreamProvider` interface. This interface is given in Display 1.

A static `WorkStream` class provides a single point of contact for the `IWorkStreamProvider` implementation object in use. The `WorkStream` static class exposes a `Provider` property, which provides get and set access to the `IWorkStreamProvider` implementation in use. If there is no `IWorkStreamProvider` implementation in use when the property value is requested, a `DefaultWorkStreamProvider` object is created and set as the `IWorkStreamProvider` in use. The `WorkStream` static class also exposes `GetNonce`,

```
/// <summary>
/// The interface that must be implemented by a work stream provider.
/// </summary>
public interface IWorkStreamProvider : IDisposable
{
    /// <summary>
    /// Get a sequence of cryptographically strong random bytes. The bytes are
    /// intended to be used as keys and IVs of encrypted streams.
    /// </summary>
    /// <param name="length">
    /// The length of the desired sequence of cryptographically strong random
    /// bytes.
    /// </param>
    byte [] GetNonce (int length);

    /// <summary>
    /// Get a work file name. The name is a path in a provider-specific work
    /// directory. The name will not conflict with the name of any file currently
    /// existing.
    /// </summary>
    string GetWorkFileName ();

    /// <summary>
    /// Get a work stream.
    /// </summary>
    IWorkStream GetWorkStream ();

    /// <summary>
    /// Purge any existing work files.
    /// </summary>
    void Purge ();
}
```

Display 1. The IWorkStreamProvider interface.

GetWorkFileName, GetWorkStream, and Purge methods that delegate to the IWorkStreamProvider implementation object in use. When the IWorkStreamProvider implementation object in use is changed, the replaced implementation object is disposed. This destroys the directory in which it placed files, and the files themselves.

The DefaultWorkStreamProvider class implements the IWorkStreamProvider interface. This class creates an RNGCryptoServiceProvider instance to provide nonces, and uses nonces to create random file names. It provides work file names in a randomly named subdirectory of the user's temporary file directory. Both the randomly named directory and the random file names are 8.3 names using numeric and lowercase alphabetic ASCII characters. This arrangement provides approximately 56.87 bits of entropy in the file names.

IWorkStreamProvider implementations produce IWorkStream implementation objects. The IWorkStream interface is given in Display 2.

The DefaultWorkStreamProvider implementation of the IWorkStreamProvider interface produces DefaultWorkStreamImplementation instances which implement the IWorkStream interface. The DefaultWorkStreamImplementation class implements work streams as CryptoStream using a TripleDESCryptoServiceProvider as the source for encryptors and decryptors, and using a DeflateStream to provide the cleartext to or to accept the cleartext from the CryptoStream. Each time a writer is produced, any previous backing file is deleted, the triple DES provider is rekeyed with an initialization vector and key nonces provided by the IWorkStreamProvider implementation, and a new backing file with a new file name provided by the IWorkStreamProvider implementation is used. In this manner, a single IWorkStream instance can be reused for serially created and destroyed work streams and repeat neither file names nor key material.

The WorkFile static class and the DefaultWorkStreamProvider implementation of the IWorkStreamProvider interface provide IWorkStream implementations on demand. For non-work stream temporary files, these classes also provide the GetFileName and GetNonce methods. These methods can be used to generate unique program-duration file names in the same directory as work stream backing files, and encryption keys for other file access methods. For the particular case of databases using connection strings that may contain a password from which an encryption key is derived, the GetNonce method can be used to acquire a reasonably-sized random byte sequence (of perhaps 16 bytes) and the Convert.ToBase64String method used to convert this to a character string password and encryption key.

```
/// <summary>
/// The interface implemented by a work stream.
/// </summary>
public interface IWorkStream : IDisposable
{
    /// <summary>
    /// Get a Stream that can read the contents written by the most recent previous
    /// writer. Any number or readers may exist simultaneously, but no readers may
    /// coexist with any writer of the same work stream.
    /// </summary>
    Stream GetReader ();

    /// <summary>
    /// Get a Stream that can write. The stream can be read only by a reader
    /// produced after the writer was produced. If a writer exists, neither a
    /// reader nor another writer can exist at the same time.
    /// </summary>
    Stream GetWriter ();
}
```

Display 2. The IWorkStream interface.

Test Program

The code associated with this Application Note includes a test program, `Program`. The test program creates a work stream, writes lines containing the numbers 0 to 9999 expressed as decimal digit strings to it, then reads the data back and ensures it is as expected. This tests the fidelity of the work stream contents: the data written is the data read back. In addition, the test program sets up a `FileSystemWatcher` object observing the user temporary directory, and reports on file system activity there. The report shows the file system activity associated with the temporary stream creation and destruction.

Running this test program on Linux and on Windows, and comparing the output, is instructive. The program output on Windows in given in Display 3; the program output on Linux is given in Display 4.

The two runs show the system-dependent user temporary directories (`C:\Users\`*`username`*`\AppData\Local\Temp` on Windows, `/tmp` on Linux), the run-dependent temporary directory names created within the user temporary directory (`4g97umvf.w3z` and `cspbb80q.m51`), and the run-dependent file name of the temporary stream backing file (`5tvnmurd.oxe` and `a7xl8pn6.03x`). The two runs also show fascinating differences in the events generated by the `FileSystemWatcher` on the two systems. On both Windows and Linux, both creation and deletion of both directories and files are captured. On Windows, the creation and deletion of a file in a directory is noted as a change in the directory itself, as is updating the length of a file in the directory (which occurs when the file is closed after writing). On Linux, none of these activities are regarded as changing the directory itself. On Windows, a single change notification is published for the work stream file; on Linux, a number of change notifications are published for the work stream file. The difference in change notifications between the platforms might be due to differences in buffering and caching. On both platforms, the event notification message and the message from the test program describing its activities race one another to the console; unsurprisingly, the order of these messages might change from run to run and platform to platform.

Summary

This Application Note has described the use of temporary files, and in particular the use of temporary sequential-only access files which it terms temporary streams. It has described some simple security considerations in the use of temporary files and streams. It has described the design and implementation of a temporary stream facility which directly addresses tempo-

```
Program: WorkStream tester
FileSystemWatcher set up on C:\Users\Brian\AppData\Local\Temp.
Created: C:\Users\Brian\AppData\Local\Temp\4g97umvf.w3z.
Work stream obtained.
Created: C:\Users\Brian\AppData\Local\Temp\4g97umvf.w3z\5tvnmurd.oxe.
Changed: C:\Users\Brian\AppData\Local\Temp\4g97umvf.w3z.
Writer started.
Changed: C:\Users\Brian\AppData\Local\Temp\4g97umvf.w3z\5tvnmurd.oxe.
Writer closed.
Reader started.
Reader closed.
Sleeping for 5 seconds.
Deleted: C:\Users\Brian\AppData\Local\Temp\4g97umvf.w3z\5tvnmurd.oxe.
WorkStream discarded.
Data was successfully written to and read back from work stream.
Cleaning up.
Changed: C:\Users\Brian\AppData\Local\Temp\4g97umvf.w3z.
Deleted: C:\Users\Brian\AppData\Local\Temp\4g97umvf.w3z.
FileSystemWatcher torn down.
Done.
```

Display 3. Test program output (Windows host).

rary streams and can be used in conjunction with data storage facilities such a file-oriented databases for more general temporary files. It has demonstrated the use of this facility, and discussed the results of instrumenting the facility.

The temporary stream facility described is useful and secure against typical unsophisticated attacks by non-privileged hostile programs, with the exception of hostile locking of the temporary stream backing file. The exception is required for multiple readers of the temporary stream to exist simultaneously. Should multiple simultaneous readers not be required, obvious modifications can prevent the hostile locking attack.

Further Information

This Application Note and a MonoDevelop and Visual Studio solution containing all source code for the temporary stream facility and sample application are contained in a .ZIP file at: http://brianhetrick.com/an/AN-10.zip.

The MonoDevelop software used to produce the code described in this Application Note is available for no charge at: http://www.monodevelop.com/.

The Raspbian operating system on which this software was produced and on which this Application Note was produced is available for no charge at: https://www.raspberrypi.org/.

The LibreOffice office suite used to create this Application Note itself is available for no charge at: http://www.libreoffice.org/.

This Application Note uses the Book Antiqua font for body text, the Source Sans Pro fonts for headings, headers, and footers, and the Source Code Pro font for computer code. The Book Antiqua font can be licensed from Monotype at: http://catalog.monotype.com/family/monotype/book-antiqua and is included in several of Microsoft's products. The Source Sans Pro font is available at no charge at: https://adobe-fonts.github.io/source-sanspro. The Source Code Pro font is available at no charge at: https://github.com/adobe-fonts/sourcecode-pro.

Disclaimer

The author believes all content of this Application Note and any software and design it describes is a

```
Program: WorkStream tester
FileSystemWatcher set up on /tmp.
Work stream obtained.
Created: /tmp/cspbb80q.m51.
Created: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Writer started.
Changed: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Changed: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Changed: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Changed: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Changed: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Changed: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Changed: /tmp/cspbb80q.m51/a7xl8pn6.03x.
Writer closed.
Reader started.
Reader closed.
Sleeping for 5 seconds.
Deleted: /tmp/cspbb80q.m51/a7xl8pn6.03x.
WorkStream discarded.
Data was successfully written to and read back from work stream.
Cleaning up.
Deleted: /tmp/cspbb80q.m51.
FileSystemWatcher torn down.
Done.
```

Display 4. Test program output (Linux host).

straightforward application of existing well-known concepts and techniques and is evident to anyone versed in the state of the art. Despite this, the author makes no claim or warranty of any kind that any item developed based on this Application Note, the software it describes, the designs it describes, or any portion of any or all, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country.

Copyright and Grant of License