SafeHandle Class

Namespace: System.Runtime.InteropServices

Assembly: System.Runtime.dll

Represents a wrapper class for operating system handles. This class must be inherited.

In this article

C#

```csharp
public abstract class SafeHandle :
System.Runtime.ConstrainedExecution.CriticalFinalizerObject,
IDisposable
```

Inheritance Object → CriticalFinalizerObject → SafeHandle

Derived Microsoft.Win32.SafeHandles.SafeAccessTokenHandle
Microsoft.Win32.SafeHandles.SafeHandleMinusOneIsInvalid
Microsoft.Win32.SafeHandles.SafeHandleZeroOrMinusOneIsInvalid
System.Security.Cryptography.SafeEvpPKeyHandle

Implements IDisposable

Examples

The following code example creates a custom safe handle for an operating system file handle, deriving from SafeHandleZeroOrMinusOneIsInvalid. It reads bytes from a file and displays their hexadecimal values. It also contains a fault testing harness that causes the thread to abort, but the handle value is freed. When using an IntPtr to represent handles, the handle is occasionally leaked due to the asynchronous thread abort.

You will need a text file in the same folder as the compiled application. Assuming that you name the application "HexViewer", the command line usage is:

```
HexViewer <filename> -Fault
```

Optionally specify `-Fault` to intentionally attempt to leak the handle by aborting the thread in a certain window. Use the Windows Perfmon.exe tool to monitor handle counts while injecting faults.

C# Copy

```
using System;
using System.Runtime.InteropServices;
using System.IO;
using System.ComponentModel;
using System.Security.Permissions;
using System.Security;
using System.Threading;
using Microsoft.Win32.SafeHandles;
using System.Runtime.ConstrainedExecution;

namespace SafeHandleDemo
{
    [SecurityPermission(SecurityAction.InheritanceDemand, UnmanagedCode
= true)]
    [SecurityPermission(SecurityAction.Demand, UnmanagedCode = true)]
    internal class MySafeFileHandle : SafeHandleZeroOrMinusOneIsInvalid
    {
        // Create a SafeHandle, informing the base class
        // that this SafeHandle instance "owns" the handle,
        // and therefore SafeHandle should call
        // our ReleaseHandle method when the SafeHandle
        // is no longer in use.
        private MySafeFileHandle()
            : base(true)
        {
        }
        [ReliabilityContract(Consistency.WillNotCorruptState,
Cer.MayFail)]
        override protected bool ReleaseHandle()
        {
            // Here, we must obey all rules for constrained execution
regions.
            return NativeMethods.CloseHandle(handle);
            // If ReleaseHandle failed, it can be reported via the
            // "releaseHandleFailed" managed debugging assistant (MDA).
This
            // MDA is disabled by default, but can be enabled in a de-
bugger
            // or during testing to diagnose handle corruption
problems.
            // We do not throw an exception because most code could not
recover
            // from the problem.
        }
    }

    [SuppressUnmanagedCodeSecurity()]
    internal static class NativeMethods
    {
        // Win32 constants for accessing files.
        internal const int GENERIC_READ = unchecked((int)0x80000000);

        // Allocate a file object in the kernel, then return a handle
to it.
        [DllImport("kernel32", SetLastError = true, CharSet =
CharSet.Unicode)]
        internal extern static MySafeFileHandle CreateFile(String
fileName,
           int dwDesiredAccess, System.IO.FileShare dwShareMode,
           IntPtr securityAttrs_MustBeZero, System.IO.FileMode
dwCreationDisposition,
           int dwFlagsAndAttributes, IntPtr hTemplateFile_MustBeZero);

        // Use the file handle.
        [DllImport("kernel32", SetLastError = true)]
        internal extern static int ReadFile(MySafeFileHandle handle,
byte[] bytes,
           int numBytesToRead, out int numBytesRead, IntPtr
overlapped_MustBeZero);

        // Free the kernel's file object (close the file).
        [DllImport("kernel32", SetLastError = true)]
        [ReliabilityContract(Consistency.WillNotCorruptState,
Cer.MayFail)]
        internal extern static bool CloseHandle(IntPtr handle);
    }

    // The MyFileReader class is a sample class that accesses an oper-
ating system
    // resource and implements IDisposable. This is useful to show the
types of
    // transformation required to make your resource wrapping classes
    // more resilient. Note the Dispose and Finalize implementations.
    // Consider this a simulation of System.IO.FileStream.
    public class MyFileReader : IDisposable
    {
        // _handle is set to null to indicate disposal of this
instance.
        private MySafeFileHandle _handle;

        public MyFileReader(String fileName)
        {
            // Security permission check.
            String fullPath = Path.GetFullPath(fileName);
            new FileIOPermission(FileIOPermissionAccess.Read,
fullPath).Demand();

            // Open a file, and save its handle in _handle.
            // Note that the most optimized code turns into two proces-
sor
            // instructions: 1) a call, and 2) moving the return value
into
            // the _handle field.  With SafeHandle, the CLR's platform
invoke
            // marshaling layer will store the handle into the SafeHan-
dle
            // object in an atomic fashion. There is still the problem
            // that the SafeHandle object may not be stored in _handle,
but
            // the real operating system handle value has been safely
stored
            // in a critical finalizable object, ensuring against leak-
ing
            // the handle even if there is an asynchronous exception.

            MySafeFileHandle tmpHandle;
            tmpHandle = NativeMethods.CreateFile(fileName,
NativeMethods.GENERIC_READ,
                FileShare.Read, IntPtr.Zero, FileMode.Open, 0,
IntPtr.Zero);

            // An async exception here will cause us to run our final-
izer with
            // a null _handle, but MySafeFileHandle's ReleaseHandle
code will
            // be invoked to free the handle.

            // This call to Sleep, run from the fault injection code in
Main,
            // will help trigger a race. But it will not cause a handle
leak
            // because the handle is already stored in a SafeHandle
instance.
```

```
            // Critical finalization then guarantees that freeing the
handle,
            // even during an unexpected AppDomain unload.
            Thread.Sleep(500);
            _handle = tmpHandle;  // Makes _handle point to a critical
finalizable object.

            // Determine if file is opened successfully.
            if (_handle.IsInvalid)
                throw new Win32Exception(Marshal.GetLastWin32Error(),
fileName);
        }

        public void Dispose()  // Follow the Dispose pattern - public
nonvirtual.
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        // No finalizer is needed. The finalizer on SafeHandle
        // will clean up the MySafeFileHandle instance,
        // if it hasn't already been disposed.
        // Howerver, there may be a need for a subclass to
        // introduce a finalizer, so Dispose is properly implemented
here.
        [SecurityPermission(SecurityAction.Demand, UnmanagedCode =
true)]
        protected virtual void Dispose(bool disposing)
        {
            // Note there are three interesting states here:
            // 1) CreateFile failed, _handle contains an invalid handle
            // 2) We called Dispose already, _handle is closed.
            // 3) _handle is null, due to an async exception before
            //    calling CreateFile. Note that the finalizer runs
            //    if the constructor fails.
            if (_handle != null && !_handle.IsInvalid)
            {
                // Free the handle
                _handle.Dispose();
            }
            // SafeHandle records the fact that we've called Dispose.
        }

        [SecurityPermission(SecurityAction.Demand, UnmanagedCode =
true)]
        public byte[] ReadContents(int length)
        {
            if (_handle.IsInvalid)  // Is the handle disposed?
                throw new ObjectDisposedException("FileReader is
closed");
```

```
            // This sample code will not work for all files.
            byte[] bytes = new byte[length];
            int numRead = 0;
            int r = NativeMethods.ReadFile(_handle, bytes, length, out
numRead, IntPtr.Zero);
            // Since we removed MyFileReader's finalizer, we no longer
need to
            // call GC.KeepAlive here.  Platform invoke will keep the
SafeHandle
            // instance alive for the duration of the call.
            if (r == 0)
                throw new Win32Exception(Marshal.GetLastWin32Error());
            if (numRead < length)
            {
                byte[] newBytes = new byte[numRead];
                Array.Copy(bytes, newBytes, numRead);
                bytes = newBytes;
            }
            return bytes;
        }
    }

    static class Program
    {
        // Testing harness that injects faults.
        private static bool _printToConsole = false;
        private static bool _workerStarted = false;

        private static void Usage()
        {
            Console.WriteLine("Usage:");
            // Assumes that application is named HexViwer"
            Console.WriteLine("HexViewer <fileName> [-fault]");
            Console.WriteLine(" -fault Runs hex viewer repeatedly, in-
jecting faults.");
        }

        private static void ViewInHex(Object fileName)
        {
            _workerStarted = true;
            byte[] bytes;
            using (MyFileReader reader = new
MyFileReader((String)fileName))
            {
                bytes = reader.ReadContents(20);
            }  // Using block calls Dispose() for us here.

            if (_printToConsole)
            {
                // Print up to 20 bytes.
                int printNBytes = Math.Min(20, bytes.Length);
                Console.WriteLine("First {0} bytes of {1} in hex",
printNBytes, fileName);
                for (int i = 0; i < printNBytes; i++)
                    Console.Write("{0:x} ", bytes[i]);
                Console.WriteLine();
            }
        }

        static void Main(string[] args)
        {
            if (args.Length == 0 || args.Length > 2 ||
                args[0] == "-?" || args[0] == "/?")
            {
                Usage();
                return;
            }

            String fileName = args[0];
            bool injectFaultMode = args.Length > 1;
            if (!injectFaultMode)
            {
                _printToConsole = true;
                ViewInHex(fileName);
            }
            else
            {
                Console.WriteLine("Injecting faults - watch handle
count in perfmon (press Ctrl-C when done)");
                int numIterations = 0;
                while (true)
                {
                    _workerStarted = false;
                    Thread t = new Thread(new
ParameterizedThreadStart(ViewInHex));
                    t.Start(fileName);
                    Thread.Sleep(1);
                    while (!_workerStarted)
                    {
                        Thread.Sleep(0);
                    }
                    t.Abort();  // Normal applications should not do
this.
                    numIterations++;
                    if (numIterations % 10 == 0)
                        GC.Collect();
                    if (numIterations % 10000 == 0)
                        Console.WriteLine(numIterations);
                }
            }
        }
        }
    }
```

Remarks

The SafeHandle class provides critical finalization of handle resources, preventing handles from being reclaimed prematurely by garbage collection and from being recycled by Windows to reference unintended unmanaged objects.

This topic includes the following sections:

Why SafeHandle?

Before the .NET Framework version 2.0, all operating system handles could only be encapsulated in the IntPtr managed wrapper object. While this was a convenient way to interoperate with native code, handles could be leaked by asynchronous exceptions, such as a thread aborting unexpectedly or a stack overflow. These asynchronous exceptions are an obstacle to cleaning up operating system resources, and they can occur almost anywhere in your app.

Although overrides to the Object.Finalize method allow cleanup of unmanaged resources when an object is being garbage collected, in some circumstances, finalizable objects can be reclaimed by garbage collection while executing a method within a platform invoke call. If a finalizer frees the handle passed to that platform invoke call, it could lead to handle corruption. The handle could also be reclaimed while your method is blocked during a platform invoke call, such as while reading a file.

More critically, because Windows aggressively recycles handles, a handle could be recycled and point to another resource that might contain sensitive data. This is known as a recycle attack and can potentially corrupt data and be a security threat.

What SafeHandle does

The SafeHandle class simplifies several of these object lifetime issues, and is integrated with platform invoke so that operating system resources are not leaked. The SafeHandle class resolves object lifetime issues by assigning and releasing handles without interruption. It contains a critical finalizer that ensures that the handle is closed and is guaranteed to run during unexpected AppDomain unloads, even in cases when the platform invoke call is assumed to be in a corrupted state.

Because SafeHandle inherits from CriticalFinalizerObject, all the noncritical finalizers are called before any of the critical finalizers. The finalizers are called on objects that are no longer live during the same garbage collection pass. For example, a FileStream object can run a normal finalizer to flush out existing buffered data without the risk of the handle being leaked or recycled. This very weak ordering between critical and noncritical finalizers is not intended for general use. It exists primarily to assist in the migration of existing libraries by allowing those libraries to use SafeHandle without altering their semantics. Additionally, the critical finalizer and anything it calls, such as the SafeHandle.ReleaseHandle() method, must be in a constrained execution region. This imposes constraints on what code can be written within the finalizer's call graph.

Platform invoke operations automatically increment the reference count of handles encapsulated by a SafeHandle and decrement them upon completion. This ensures that the handle will not be recycled or closed unexpectedly.

You can specify ownership of the underlying handle when constructing SafeHandle objects by supplying a value to the `ownsHandle` argument in the SafeHandle class constructor. This controls whether the SafeHandle object will release the handle after the object has been disposed. This is useful for handles with peculiar lifetime requirements or for consuming a handle whose lifetime is controlled by someone else.

Classes derived from SafeHandle

SafeHandle is an abstract wrapper class for operating system handles. Deriving from this class is difficult. Instead, use the derived classes in the Microsoft.Win32.SafeHandles namespace that provide safe handles for the following:

- Files (the SafeFileHandle class).
- Memory mapped files (the SafeMemoryMappedFileHandle class).
- Pipes (the SafePipeHandle class).
- Memory views (the SafeMemoryMappedViewHandle class).

- Cryptography constructs (the SafeNCryptHandle, SafeNCryptKeyHandle, SafeNCryptProviderHandle, and SafeNCryptSecretHandle classes).
- Processes (the SafeProcessHandle class).
- Registry keys (the SafeRegistryHandle class).
- Wait handles (the SafeWaitHandle class).

Notes to Implementers

To create a class derived from SafeHandle, you must know how to create and free an operating system handle. This process is different for different handle types because some use the CloseHandle function, while others use more specific functions such as UnmapViewOfFile or FindClose. For this reason, you must create a derived class of SafeHandle for each operating system handle type that you want to wrap in a safe handle.

When you inherit from SafeHandle, you must override the following members: IsInvalid and ReleaseHandle().

You should also provide a parameterless constructor that calls the base constructor with a value that represent an invalid handle value, and a Boolean value indicating whether the native handle is owned by the SafeHandle and consequently should be freed when that SafeHandle has been disposed.

Constructors

SafeHandle(IntPtr, Boolean)	Initializes a new instance of the SafeHandle class with the specified invalid handle value.

Fields

handle	Specifies the handle to be wrapped.

Properties

IsClosed	Gets a value indicating whether the handle is closed.

IsInvalid	When overridden in a derived class, gets a value indicating whether the handle value is invalid.

Methods

Close()	Marks the handle for releasing and freeing resources.
DangerousAddRef(Boolean)	Manually increments the reference counter on SafeHandle instances.
DangerousGetHandle()	Returns the value of the handle field.
DangerousRelease()	Manually decrements the reference counter on a SafeHandle instance.
Dispose()	Releases all resources used by the SafeHandle class.
Dispose(Boolean)	Releases the unmanaged resources used by the SafeHandle class specifying whether to perform a normal dispose operation.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
Finalize()	Frees all resources associated with the handle.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
ReleaseHandle()	When overridden in a derived class, executes the code required to free the handle.
SetHandle(IntPtr)	Sets the handle to the specified pre-existing handle.
SetHandleAsInvalid()	Marks a handle as no longer used.

ToString()	Returns a string that represents the current object. (Inherited from Object)

Applies to

.NET

5.0 RC1

.NET Core

3.1, 3.0, 2.2, 2.1, 2.0, 1.1, 1.0

.NET Framework

4.8, 4.7.2, 4.7.1, 4.7, 4.6.2, 4.6.1, 4.6, 4.5.2, 4.5.1, 4.5, 4.0, 3.5, 3.0, 2.0

.NET Standard

2.1, 2.0, 1.6, 1.5, 1.4, 1.3, 1.2, 1.1

UWP

10.0

Xamarin.Android

7.1

Xamarin.iOS

10.8

Xamarin.Mac

3.0

See also

- Microsoft.Win32.SafeHandles
- CriticalHandle
- CriticalFinalizerObject