1 Continuing Detours: the reinvention of Windows API Hooking

Microsoft® Detours latest release was in December 2006. Now times have changed and the NET Framework has become more and more popular. Besides the well known unmanaged code hooking, EasyHook provides a way to hook unmanaged code from a managed environment. This implies several advantages:

- **No resource or memory leaks** are left in the target
- You can write pure managed hook handlers for unmanaged APIs
- All hooks are installed and **automatically removed** in a stable manner
- You can use all the convenience managed code provides, like **NET Remoting, WCF and WPF**
- You will be able to write injection libraries and host processes compiled for ***AnyCPU,*** which will allow you to inject your code into 32- and 64-Bit processes from 64- and 32-Bit processes by using the very same assembly in all cases.

This way hooking has become a simple task and you can now write hooking applications like FileMon or RegMon with a few lines of code.

Further EasyHook 2.5 provides additional features like:

- Experimental stealth injection for unmanaged code not raising attention of any current AV
- 32- and 64-Bit Kernel mode hooking support, since Windows XP.
- A pure unmanaged hooking core which will improve performance, stability and compatibility.
- A solid unmanaged API for writing hooking apps and libraries without the NET Framework
- The unmanaged core does not require CRT bindings and thus will reduce deployment size about some megabytes. Also Windows 2000 SP4 and Windows Server 2008 SP1 can now be targeted with the same EasyHook binary.

Minimal software requirements for end-users to execute applications using EasyHook:

- Windows 2000 SP4 or later
- Microsoft NET Framework 2.0 Redistributable

Table of Content

ATTENTION

This Guide will cover the managed part of EasyHook only. Most things also apply to the unmanaged API. Refer to the “Unmanaged API Reference” for more information. The “Managed API Reference” also contains much additional information to the stuff covered here.

LICENSE CHANGE

EasyHook is now released under the Lesser GPL instead of the MIT License.

1.1 Security Advisor

Unlike what some (commercial) hooking libraries out there are advertising to boost sales, user-mode hooking can NEVER be an option to apply additional security checks in any safe manner. If you only want to "sandbox" a dedicated process, you know well about, and the process in fact doesn't know about EasyHook, this might succeed! But don't ever attempt to write any security software based on user mode hooking. It won't work, I promise you… This is also why EasyHook does not support a so called "System wide" injection, which in fact is just an illusion, because as I said, with user-mode hooks this will always be impossible. But if you want to keep this illusion you may stick with other (commercial) libraries attempting to do so…

Since EasyHook 2.5, you are able to easily hook 32-Bit kernels. Even if EasyHook would allow hooking 64-Bit kernels, I don't recommend this because then you would get trouble with PatchGuard. Bypassing PatchGuard is possible, at least these days, but the chance of BSODing your customer's PCs is too big. You should consider purchasing the PatchGuard API which will allow you to write security apps based on kernel mode interceptions. Kernel mode hooking (or the PatchGuard API) is the only option to apply additional security checks. Since Windows Vista, also the Windows Filtering Platform and other Vista specific APIs will be helpful to write security software!

So what is user-mode hooking for? In general, user-mode hooking is intended for API monitoring, like Mark Russinovich's ProcessMonitor (alias FileMon/RegMon), resource leak detection, various malware which doesn't need to care about security issues, extending applications and libraries you don't have the source code for (also cracks may fall in this category), adding a compatibility layer for existing applications to run on newer OSes, etc.

If anyone uses security in context of user-mode hooks, your alarm bells should ring!

1.2 A simple FileMon derivate

To prove that EasyHook really makes hooking simple, look at the following demo application, which will log all file accesses from a given process. We need a host process which injects the library and displays file accesses. It is possible to combine injection library and host process in one file as both are just threaded as valid NET assemblies, but I think to separate them is a more consistent approach. This demo will be used throughout the whole guide:

```
using System;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Text;
using EasyHook;
```

```
namespace FileMon
{
    public class FileMonInterface : MarshalByRefObject
    {
        public void IsInstalled(Int32 InClientPID)
        {
            Console.WriteLine("FileMon has been installed in target
{0}.\r\n", InClientPID);
        }

        public void OnCreateFile(Int32 InClientPID, String[] InFileNames)
        {
            for (int i = 0; i < InFileNames.Length; i++)
            {
                Console.WriteLine(InFileNames[i]);
            }
        }

        public void ReportException(Exception InInfo)
        {
            Console.WriteLine("The target process has reported an error:\r\n"
+ InInfo.ToString());
        }

        public void Ping()
        {
        }
    }

    class Program
    {
        static String ChannelName = null;

        static void Main(string[] args)
        {
            try
            {
                Config.Register(
                    "A FileMon like demo application.",
                    "FileMon.exe",
                    "FileMonInject.dll");

                RemoteHooking.IpcCreateServer<FileMonInterface>(ref
ChannelName, WellKnownObjectMode.SingleCall);

                RemoteHooking.Inject(
                    Int32.Parse(args[0]),
                    "FileMonInject.dll",
                    "FileMonInject.dll",
                    ChannelName);

                Console.ReadLine();
            }
            catch (Exception ExtInfo)
            {
                Console.WriteLine("There was an error while connecting to
target:\r\n{0}", ExtInfo.ToString());
```

```
            }
        }
    }
}
```

The most complex part is the injected library which has to fulfill various requirements. We are hooking the CreateFile-API and redirecting all requests to our host process. The library will be unloaded if the host process is terminated:

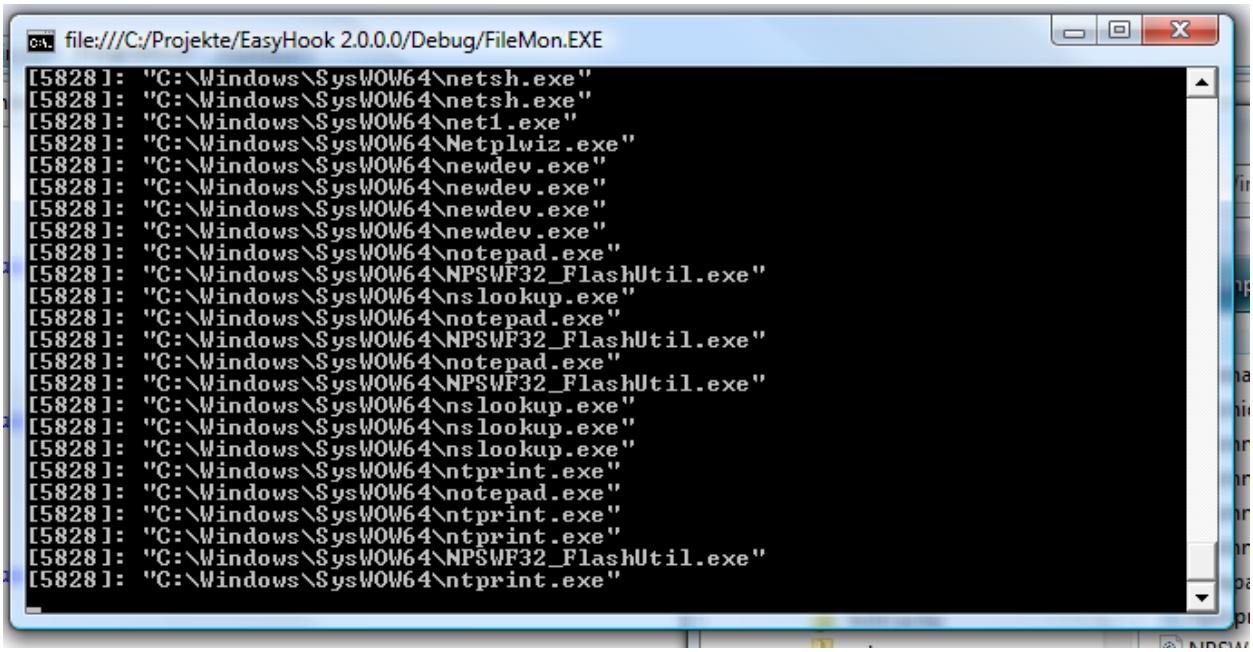
```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Runtime.InteropServices;
using EasyHook;

namespace FileMonInject
{
    public class Main : EasyHook.IEntryPoint
    {
        FileMon.FileMonInterface Interface;
        LocalHook CreateFileHook;
        Stack<String> Queue = new Stack<String>();

        public Main(
            RemoteHooking.IContext InContext,
            String InChannelName)
        {
            // connect to host...

            Interface =
RemoteHooking.IpcConnectClient<FileMon.FileMonInterface>(InChannelName);
        }

        public void Run(
            RemoteHooking.IContext InContext,
            String InChannelName)
        {
            // install hook...
            try
            {
                CreateFileHook = LocalHook.Create(
                    LocalHook.GetProcAddress("kernel32.dll", "CreateFileW"),
                    new DCreateFile(CreateFile_Hooked),
                    this);

                CreateFileHook.ThreadACL.SetExclusiveACL(new Int32[] { 0 });
            }
            catch (Exception ExtInfo)
            {
```

```
                Interface.ReportException(ExtInfo);

                return;
            }

            Interface.IsInstalled(RemoteHooking.GetCurrentProcessId());

            // wait for host process termination...
            try
            {
                while (true)
                {
                    Thread.Sleep(500);

                    // transmit newly monitored file accesses...
                    if (Queue.Count > 0)
                    {
                        String[] Package = null;

                        lock (Queue)
                        {
                            Package = Queue.ToArray();

                            Queue.Clear();
                        }


Interface.OnCreateFile(RemoteHooking.GetCurrentProcessId(), Package);
                    }
                    else
                        Interface.Ping();
                }
            }
            catch
            {
                // NET Remoting will raise an exception if host is
unreachable
            }
        }

        [UnmanagedFunctionPointer(CallingConvention.StdCall,
            CharSet = CharSet.Unicode,
            SetLastError = true)]
        delegate IntPtr DCreateFile(
            String InFileName,
            UInt32 InDesiredAccess,
            UInt32 InShareMode,
            IntPtr InSecurityAttributes,
            UInt32 InCreationDisposition,
            UInt32 InFlagsAndAttributes,
            IntPtr InTemplateFile);

        // just use a P-Invoke implementation to get native API access from
C# (this step is not necessary for C++.NET)
        [DllImport("kernel32.dll",
            CharSet = CharSet.Unicode,
            SetLastError = true,
```

```
            CallingConvention = CallingConvention.StdCall)]
        static extern IntPtr CreateFile(
            String InFileName,
            UInt32 InDesiredAccess,
            UInt32 InShareMode,
            IntPtr InSecurityAttributes,
            UInt32 InCreationDisposition,
            UInt32 InFlagsAndAttributes,
            IntPtr InTemplateFile);

        // this is where we are intercepting all file accesses!
        static IntPtr CreateFile_Hooked(
            String InFileName,
            UInt32 InDesiredAccess,
            UInt32 InShareMode,
            IntPtr InSecurityAttributes,
            UInt32 InCreationDisposition,
            UInt32 InFlagsAndAttributes,
            IntPtr InTemplateFile)
        {
            try
            {
                Main This = (Main)HookRuntimeInfo.Callback;

                lock (This.Queue)
                {
                    This.Queue.Push(InFileName);
                }
            }
            catch
            {
            }

            // call original API...
            return CreateFile(
                InFileName,
                InDesiredAccess,
                InShareMode,
                InSecurityAttributes,
                InCreationDisposition,
                InFlagsAndAttributes,
                InTemplateFile);
        }
    }
}
```

Even if this might look strange, the next chapters will explain what is done there and why. You may start this application with a user defined target process ID as one and only parameter from command line. I recommend using the PID of “explorer.exe” because this will immediately produce output! Just browse your file system while running the FileMon utility:

```
Command line utility->   FileMon.exe %PID%
```



It is also possible to output the whole thing into a file what might provide more convenience:

Command line utility-> FileMon.exe %PID% > “C:\MyLog.txt”

2 A deep look under the hook

Now that you have seen the basic ideas of EasyHook and some sample code, we should start to discover what is really going on under the hood. In this chapter you will learn how to utilize most parts of the EasyHook API, injecting libraries into any process and hooking any API you want.

2.1 Global Assembly Cache

Currently EasyHook is expecting every injected assembly including all of its dependencies in the Global Assembly Cache (GAC). This is because the CLR will only search for assemblies in directories relative to the current application base directory and the GAC and therefore a target process normally has no access to EasyHook or your injected library. EasyHook is using a reference counter to make sure that multiple installations of the same assemblies from different applications can be managed. The following will register all EasyHook components and the two user assemblies required for injection in the GAC.

The first parameter is just an uninterpreted string which should describe what your service is doing. All further parameters are expected to be relative/absolute file paths referring to all assemblies that should be temporarily registered in GAC. Please note that only strongly named assemblies are accepted.

```
Config.Register(
        "A FileMon like demo application.",
        "FileMon.exe",
        "FileMonInject.dll");
```

It is guaranteed that your libraries will be removed from GAC if the injecting process is being terminated in all common cases. Of course there are some rare exceptions, for example if you shutdown your PC by disconnecting the power cable. In such a case the assemblies will remain in the GAC, forever, which is no bad thing in end-user scenarios but truly during development. You may use the ***Gacutil.exe*** that ships with Visual Studio to remove all temporary GAC assemblies.

1) Open the “Visual Studio Command Promt” as administrator.
2) Run the commands:
   gacutil /uf EasyHook
3) Run additional commands for each of your assemblies that should be removed from the GAC…

2.2 Windows Defender

Injection will sometimes make the Windows Defender complain:

Windows Defender Real-Time Protection agent has detected changes. Microsoft recommends you analyze the software that made these changes for potential risks. You can use information about how these programs operate to choose whether to allow them to run or remove them from your computer. Allow changes only if you trust the program or the software publisher. Windows Defender can't undo changes that you allow.
For more information please see the following:

Not Applicable
Scan ID: {44726E79-4262-454E-AFED-51A30D34BF67}
User: Lynn-PC\Lynn
Name: Unknown
ID:
Severity ID:
Category ID:

```
        Path Found: process:pid:864;service:EasyHook64Svc;file:D:\Projects\EasyHook
2.0.0.0\Debug\x64\EasyHook64Svc.exe
        Alert Type: Unclassified software
        Detection Type:
```

Such warnings are immediately followed by information pointing out that Windows Defender has prevented a malicious attempt. I think this will vanish if you sign all executable binaries of EasyHook with AuthentiCode. Such blocking only occurs when injecting into essential system services.

2.3 Injection – A burden made easy

In general, library injection is one of the most complicated parts of any hooking library. But EasyHook goes further. It provides three layers of injection abstraction and your library is the fourth one. The first layer is pure, relocatable assembler code. It launches the second layer, an unmanaged C++ method. The assembler code itself is really stable. It provides extensive error information and is able to unload itself without leaving any resource leaks in the target. The C++ layer starts the managed injection loader and adjusts the target's PATH variable by adding the injecting process' application base directory as first entry. This way you will have access to any file you would also have access to from your injecting process. The managed injection loader uses NET Reflection and NET Remoting to provide extensive error reports in case of failure and to find a proper entry point in your injection library. It also cares about graceful hook removal and resource cleanup. It is supported to load the same library multiple times into the same target!

Another complex part is run on host side. It is supported to inject libraries into other terminal sessions, system services and even through WOW64 boundaries. To you, all cases seem the same. EasyHook will automatically select the right injection procedure. If EasyHook has succeeded injection, you can be 99% sure that your library has been successfully loaded and executed. If it fails you can be 99% sure that no resource leaks are left in the target and it remains in a stable, hookable state! Nearly all possible failures are being caught and it would be like a lottery win to see a target getting crashed by library injection!

Please note that Windows Vista has advanced security for its subsystem services. They are running in a protected environment like the "Protected Media Path". It is not possible to hook such services with EasyHook or any other user-mode library.

The following shows the API method that we are talking about:

```
RemoteHooking.Inject(
        Int32.Parse(args[0]),
        "FileMonInject.dll", // 32-Bit version
        "FileMonInject.dll", // 64-Bit version
```

```
            ChannelName);
```

The first four parameters are required. If you only want to hook either 32- or 64-Bit targets, you can set the unused path to ***null***. You may either specify a file path that EasyHook will automatically translate to a full qualified assembly name or a partial assembly name like "FileMonInject, PublicKeyToken = 3287453648abcdef". Currently there is only one injection option preventing EasyHook from attaching a debugger to the target but you should only set this option if the target does not like an attached debugger. EasyHook will detach it before injection is completed so in general there is nothing to worry about and it increases injection stability about magnitudes by using the target symbol addresses instead of assuming that the local ones remain valid in the target!

You can pass as many additional parameters as you like but be aware of that you shall only pass types that are accessible through GAC, otherwise the injected library is not able to deserialize the parameter list. In such a case the exception will be redirected to the host process and you may catch it with a try-catch statement around ***RemoteHooking.Inject()***. That's one of the great advantages!

The injected library will automatically get access to all additional parameters you specify after the fourth one. This way you can easily pass channel names to the target so that your injected library is able to connect to your host.

Attention

> Keep in mind that the CLR will unload your library only if the target is being terminated. Even if EasyHook releases all associated resources much earlier, you won't be able to change the injected DLL which implies that the corresponding GAC library is not updateable until the target is terminated. So if you need to change your injected library very frequently (during development) you should always terminate the target after each debugging session. This will ensure that no application depends on the library and it can be removed from the GAC.

2.3.1 Creating an already hooked process

Sometimes it is necessary to hook a process from the beginning. This is no big deal, just call *RemoteHooking.CreateAndInject* instead of *Inject*. This will execute your library main method before any other instruction. You can resume the newly created process by calling *RemoteHooking.WakeUpProcess* from your injected library *Run* method. This only makes sense in conjunction with *CreateAndInject*, otherwise it will do nothing.

2.4 The injected library entry point

All injected libraries have to export at least one public class implementing the *EasyHook.IEntryPoint* interface. The interface itself is empty but identifies your class as entry point. A class marked as entry point this way, is expected to export an instance constructor and a Run instance method having the signature "void Run(IContext, %ArgumentList%)" and ".ctor(IContext, %ArgumentList%)". Please note that "%ArgumentList%" is a placeholder for additional parameters passed to *RemoteHooking.Inject().* The list is starting with the fifth parameter you passed to Inject() and will be passed to both, the constructor and Run(). The list is not passed as array but as expanded parameter list. For example if you called *Inject(Target, Options, Path32, Path64, String, Int32, MemoryStream)*, then %ArgumentList% would be "String, Int32, MemoryStream" and your expected *Run()* signature "void Run(IContext, String, Int32, MemoryStream)". EasyHook enforces strict binding which means that the parameter list is not casted in any way. The types passed to *Inject* shall be exactly the same as in the *Run* signature. I hope this explains it.

The next thing to mention is that you should avoid using static fields or properties. Only if you know for sure that it is not possible having two instances of your library in the same target simultaneously, you can safely use static variables!

2.4.1 The library constructor

The constructor is called immediately after EasyHook has gained control in the target process. You should only connect to your host and validate the parameters. At this point EasyHook already has a working connection to the host so all exceptions you are leaving unhandled, will automatically be redirected to the host process. A common constructor may look like this:

```
public class Main : EasyHook.IEntryPoint
{
    FileMon.FileMonInterface Interface;
    LocalHook CreateFileHook;
    Stack<String> Queue = new Stack<String>();

    public Main(RemoteHooking.IContext InContext, String InChannelName)
    {
        // connect to host...
        Interface =
RemoteHooking.IpcConnectClient<FileMon.FileMonInterface>(InChannelName);

        // validate connection...
        Interface.Ping();
    }
}
```

2.4.2 The library Run-Method

The *Run* method can be threaded as application entry point. If you return from it, your library will be unloaded. But this is not really true ;-). In fact your library stays alive until the CLR decides to unload it. This behavior might change in future EasyHook versions by utilizing the CLR Hosting API, but currently we simply don't know about!

In contrast to the constructor, your *Run* method has no exception redirection. If you leave any exception unhandled, it will just initiate the usual unload procedure. In debug versions of EasyHook, you will find such unhandled exceptions in event logs. You should install all hooks and notify your host about success, what might look like this:

```
public void Run(RemoteHooking.IContext InContext, String InChannelName)
{
    // install hook...
    try
    {
        CreateFileHook = LocalHook.Create(
            LocalHook.GetProcAddress("kernel32.dll", "CreateFileW"),
            new DCreateFile(CreateFile_Hooked),
            this);

        CreateFileHook.ThreadACL.SetExclusiveACL(new Int32[] {0});
    }
    catch(Exception ExtInfo)
    {
        Interface.ReportException(ExtInfo);

        return;
    }

    Interface.IsInstalled(RemoteHooking.GetCurrentProcessId());

    // wait for host process termination...
    try
    {
        while (true)
        {
            Thread.Sleep(500);

            // transmit newly monitored file accesses...
            if (Queue.Count > 0)
            {
                String[] Package = null;

                lock (Queue)
                {
                    Package = Queue.ToArray();

                    Queue.Clear();
                }
```

```
                Interface.OnCreateFile(RemoteHooking.GetCurrentProcessId(),
Package);
            }
            else
                Interface.Ping();
        }
    }
    catch
    {
        // NET Remoting will raise an exception if host is unreachable
    }
}
```

The loop simply sends the currently queued files accesses to the host process. If the host process is being terminated, such attempts throw an exception which causes the CLR to return from the *Run* method and automatically unload your library!

2.5 Injection helper routines

There are several methods that you will find useful when dealing with injection.

To query if the current user is administrator, you can use ***RemoteHooking.IsAdministrator***. Please note that injection will fail in most cases if you don't have admin privileges! Vista is using the UAC for evaluating to admin privileges and so you should read some MSDN articles about how to utilize it.

If you already are admin, you may use the ***RemoteHooking.ExecuteAsService<T>()*** method to execute a given static method under system privileges without the need to start a service. This is potentially useful when enumerating running processes of all sessions or for any other information querying task, which might require highest privileges. Keep in mind that the static method will be executed within a system service. So any handle or other process related information will be invalid when transmitted back into your process. You should design such a method so that you retrieve all information and store it in a serializable, process independent form. This form shall be an object that is returned and this way sent back to your application by NET Remoting.

If you want to determine whether a target process is 64-Bit or not, you may use ***RemoteHooking.IsX64Process()***. But be aware of that you need *PROCESS_QUERY_INFORMATION* access to complete the call. It will also work on 32-Bit only Windows versions like Windows 2000, of course, by returning *false* in any case.

Further there are ***RemoteHooking.GetCurrentProcessId()*** and ***GetCurrentThreadId()*** which might help to query the real native values in a pure managed environment! Managed thread IDs don't necessarily map to native ones, when thinking about the coming NET 4.0.

2.6 How to install a hook

To install a hook you are required to pass at least two parameters. The first one is the entry point address you want to hook and the second one is the delegate where calls should be redirected to. The delegate shall have the *UnmanagedFunctionPointerAttribute* and also the exact call signature as the corresponding P-Invoke implementation. The best way is to look for a well tested P-Invoke implementation already out in the net and just make a delegate out of it. The managed hook handler also has to match this signature what is automatically enforced by the compiler… A P-Invoke implementation with the *DllImportAttribute* may be used to call the original API within the handler which will be necessary in most cases. Don't forget that most APIs are expected to *SetLastError*() in case of failure. So you should set it to *ERROR_ACCESS_DENIED* or *ERROR_INTERNAL_ERROR* for example if your code does not want to execute the call. Otherwise external code might behave unexpected!

A third parameter provides a way to associate an uninterpreted callback object with the hook. This is exactly the object accessible through *HookRuntimeInfo.Callback* later in the handler.

To uninstall a hook just remove all references to the object obtained during creation. To prevent it from being uninstalled you have to keep the reference of course… This is always a delayed removal because you won't know when a hook is finally removed and your handler is never called again. If you want to remove it immediately you have to call *LocalHook.Dispose* like known from dealing with unmanaged resources as file streams are.

The following code snipped is an example of how to install a hook that is excluding the current thread from being intercepted:

```
CreateFileHook = LocalHook.Create(
      LocalHook.GetProcAddress("kernel32.dll", "CreateFileW"),
      new DCreateFile(CreateFile_Hooked),
      this);

CreateFileHook.ThreadACL.SetExclusiveACL(new Int32[] {0});
```

EasyHook also does provide a way to install pure unmanaged hooks using *LocalHook.CreateUnmanaged*. You may write them using C++.NET that allows you to combine managed and unmanaged code. But keep in mind that you won't have access to the *HookRuntimeInformation* class. You would need to directly call the unmanaged API available since EasyHook 2.5, to access runtime information from unmanaged code.

But, all protection mechanisms (see next paragraph) will still wrap around your unmanaged code. An empty unmanaged hook is about magnitudes faster than an empty managed one. If your handler once

has gained execution, both are running with the same speed. The costly operation is the switch from unmanaged to managed environment and vice versa, which is not required when using pure unmanaged hooks! So your handler will be invoked in approx. 70 nanoseconds whereas a managed handler requires up to some microseconds… In some scenarios you will need this speed gain and this is why EasyHook offers it.

2.7 How to write a hook handler

Until now there was nothing complicated and I hope you agree. But writing a hook handler is something very strange. EasyHook already does provide several mechanisms to make writing hook handlers much easier or let's say possible at all:

- A Thread Deadlock Barrier (TDB) which will allow you and any subcalls to invoke the hooked API from within its handler again. Normally this would lead into a deadlock because the handler would invoke itself again and again. EasyHook will prevent such loops! This also provides the advantage that you don't need to keep track of a clean entry point.
- An OS loader lock protection which will prevent your handler from being executed in an OS loader lock and in case of managed handler code attempting so would crash the process!
- A Thread ACL model allowing you to exclude well known dedicated threads, used to manage your hooking library (for example threads that are communicating with your host), from being intercepted. Refer to the chapter "Guidelines for stable hooking", to learn about the differences and why the TDB is not enough!
- A mechanism to provide hook specific callbacks through a static class named *HookRuntimeInfo*. This way you are able to access the library instance without using a static variable for example. Additionally since EasyHook 2.5, you may generate call stack traces; determine the calling module, the handler return address, the address of this return address, etc.

Without some of the above mechanisms it would be simply impossible to use managed code as hook handler and this is what is unique to EasyHook. All of those mechanisms are very stable and heavily tested with hundred simultaneous threads executing hundred thousands of hooks (on a quad-core CPU). And the best thing is that all of them are also available in kernel-mode.

Using a hook handler you can simply provide your own implementation for the hooked API. But you should read and understand the related API documentation in detail, to provide the correct behavior for external code. If it is possible you should handle an interception as fast as possible and negotiate access or whatever within the injected library. Only in rare cases you should redirect calls to the host application in a synchronous manner as this will heavily slow down the hooked application; for example if an access negotiation can't be completed with the knowledge of the library. In a real world application you should queue all requests and transmit them periodically as an array and not every single call. This can be done like it is shown in the FileMon demo.

Keep in mind that if you are compiling for 64-Bit or AnyCPU, you have to use the right type replacements. For example *HANDLE* does NOT map to *Int32*, but to *IntPtr*. In case of 32-Bit this is not important but when switching to 64-Bit a handle is 64-Bit wide, like *IntPtr*. A DWORD in contrast will always be 32-Bit as its name implies.

The following is an example hook handler as used in the FileMon demo:

```
static IntPtr CreateFile_Hooked(
    String InFileName,
    UInt32 InDesiredAccess,
    UInt32 InShareMode,
    IntPtr InSecurityAttributes,
    UInt32 InCreationDisposition,
    UInt32 InFlagsAndAttributes,
    IntPtr InTemplateFile)
{
    try
    {
        Main This = (Main)HookRuntimeInfo.Callback;

        lock (This.Queue)
        {
            if (This.Queue.Count < 1000)
                  This.Queue.Push(InFileName);
        }
    }
    catch
    {
    }

    // call original API...
    return CreateFile(
        InFileName,
        InDesiredAccess,
        InShareMode,
        InSecurityAttributes,
        InCreationDisposition,
        InFlagsAndAttributes,
        InTemplateFile);
}
```

You may wonder about the queue limitation of 1000. The problem is that both demo applications were not designed to handle huge amounts of interceptions. I had Tortiose SVN installed and this caused my explorer to raise hundred thousands of pipe accesses. For my surprise, IPC and the injected library have no problems in dealing with that much data, but the ProcessMonitor blocked with 100% CPU usage while trying to add about 300.000 entries to the data grid. I suppose FileMon would react equally, because it will take a while to write 300.000 console lines. As the overall remote hooking mechanism seems to be stable even in such high performance scenarios, it just depends on your host application

whether you can handle it or not. To keep things simple, both demos are NOT able to handle high-performance scenarios!

Thus, if your host application is fast enough, you can safely remove the queue limitation.

2.8 Using Thread ACLs

EasyHook manages a global *ThreadACL* and also an ACL for every hook. Further each ACL can either be inclusive or exclusive. This allows you to compose nearly any kind of access negotiation based on thread IDs without much effort. By default EasyHook sets an empty global exclusive ACL, which will grant access for all threads, and an empty inclusive local ACL for every hook, which will finally deny access for all threads. All hooks are installed virtually suspended meaning no threads will pass access negotiation. This is to prevent hook handler invocation before you are able to initialize possible structures, like ACLs for example. To enable a hook for all threads just set its local ACL to an empty exclusive one. To enable it for the current thread only, just set a local inclusive ACL with zero as one and only entry. A thread ID of zero will automatically be replaced by the current thread ID BEFORE the ACL is set (this is negotiation will later use your thread ID and doesn't know anything about zero). The following is a pseudo-code of *IsThreadIntercepted*:

```
if(ACLContains(&Unit.GlobalACL, CheckID))
{
	if(ACLContains(LocalACL, CheckID))
	{
		if(LocalACL->IsExclusive)
			return FALSE;
	}
	else
	{
		if(!LocalACL->IsExclusive)
			return FALSE;
	}

	return !Unit.GlobalACL.IsExclusive;
}
else
{
	if(ACLContains(LocalACL, CheckID))
	{
		if(LocalACL->IsExclusive)
			return FALSE;
	}
	else
	{
		if(!LocalACL->IsExclusive)
			return FALSE;
	}
```

```
            return Unit.GlobalACL.IsExclusive;
        }
```

The code does nothing more than computing an intersection of the thread ID sets represented by the global and local ACL. An ACL always describes a set of threads that will be intercepted. The fact that an ACL may be inclusive or exclusive has no impact on this computation. It is just a way for you to make it easier to define a set of threads. An exclusive ACL start with ALL threads and you can specify threads that should be removed from the set. An inclusive ACL start with an empty thread set and you may specify this set manually.

The method returns TRUE, if the intersection of the global and local set of intercepted threads contains the given thread ID, FALSE otherwise.

Just play around with them and use *LocalHook.IsThreadIntercepted* to check whether your ACLs will provide expected access negotiation.

2.9 Using handler utilities

EasyHook exposes some debugging routines which may be extended in future versions. They are statically available through the *EasyHook.LocalHook* class. Currently they solve the following issues which are common when writing hook handlers:

- Translate a thread handle back to its thread ID (requires the handle to have *THREAD_QUERY_INFORMATION* access).
- Translate a process handle back to its process ID (requires the handle to have *PROCESS_QUERY_INFORMATION* access).
- Query kernel object name for any given handle. This way you are able to convert a file handle obtained by *CreateFile*() back to its file name. This will even work if the handle has no access to anything!

In contrast to EasyHook 2.0, the latest version does not require a debugger for any of the handler utilities. The only thing is that a debugger is still required to relocate RIP-relative addresses. By default a debugger is disabled. To get support for RIP relocation (which is in general not necessary for the Windows API), just call *LocalHook.EnableRIPRelocation* before hook installation.

Additionally EasyHook 2.5 exposes the *HookRuntimeInformation* class, which provides handler-specific support routines:

- Query *ReturnAddress*, query *AddressOfReturnAddress*, generate managed/unmanaged module stack backtrace, query calling managed/unmanaged module and the callback specified during hook creation
- Setup a proper stack frame to allow exception throwing and custom stack traces. EasyHook technically would make such a stack trace impossible, but since version 2.5 the stack image can be restored for a defined code section to get rid of this issue.

2.10 The IPC helper API

The core part of any target-host scenario is IPC. With NET remoting this has become really amazing. As you can see in the FileMon demo it is a thing of two lines to setup a stable, fast and secure IPC channel between the injected library and host. Of course this is only possible with the IPC helper routines exposed by EasyHook. Using the native *IpcChannel*s the code blows up to three A4 pages which is still quiet small. The helper routines will take care of serialization setup and channel privileges so that you can even connect a system service with a normal user application running without admin privileges. It also offers to generate a random port name. This service should always be used because it is the only way to get a connection secure! If you want to provide your own name, you also have to specify proper well known SIDs, which are allowed to access the channel. You should always specify the built in admin group in this case, because all admin users could crash the whole system so you don't have to worry about being exploited by an admin!

To create a server with a random port name, just call:

```
String ChannelName = null;

RemoteHooking.IpcCreateServer<FileMonInterface>(
            ref ChannelName,
            WellKnownObjectMode.SingleCall);
```

Pass the generated port name to the client and call:

```
FileMon.FileMonInterface Interface =
    RemoteHooking.IpcConnectClient<FileMon.FileMonInterface>(InChannelName);
```

From now on you are able to call the server by using the client instance of the returned underlying *MarshalByRefObject* and those calls will be automatically redirected to the server. Isn't that great?! But be aware of that this will only apply to instance members! Static members and fields will always be processed locally…

2.11 Guidelines for stable hooking

Even if EasyHook provides a new dimension of API hooking, there are still several things that you should know about, before starting to write your own handlers. The TDB will protect you from per thread deadlocks but during hooking you will often run into multi-thread deadlocks which are covered here.

Scenario 1

Imagine you would rewrite the FileMon demo sending the intercepted call directly to the target, without going through any asynchronous queue. Even if this might look correct, because the TDB should take care of it, there is indeed a big bug in it. The issue is not within your code itself but within the code your code is calling. And that is nearly always the source of bugs in hook handlers.

The problem is within NET Remoting. You already established a connection before hook installation and this is why the above approach, directly sending the interceptions back to the host, might initially work. But from time to time the CLR will reconnect to the host using CreateFile. This wouldn't be messy because the TDB would take care of it, but obviously NET Remoting does it in a queued worker item. This will call CreateFile in another thread while the intercepted one is waiting for it to complete. But because it is executed in another thread, it is also intercepted again. The CLR is still not connected and will start the whole procedure again and again until the worker item queue is full and no further threads are available. This is the time where the target will hang.

Of course I am only speculating because I don't know about the internals of NET Remoting, but this deadlock will occur! Someone might say that a solution would be to call the original API before sending the interception back to the host and maybe it would be, but keep in mind how easy our hook handler was and this simple deadlock scenario can easily apply to any complex one which you can't solve this way. It was just to show you the basic idea…

Scenario 2

We now have learned and would never make the above mistake again, would we? Let's say we are using a dedicated thread to send our queued interceptions to our host. The *Stack<T>* class will usually be a good choice and to make our code thread safe, we are using a lock, which we are entering once to transmit the stack content and once to add a new interception from within our hook handler.

Now the following happens. To send the result back to our host, the dedicated thread might call CreateFile again while holding the lock. If NET Remoting would use the same thread to call this API, the recursive lock capabilities of *Monitor::Enter*() and the TDB would do their part and prevent any deadlock. But sometimes NET Remoting will use another thread and as we are already holding the lock it could never enter it. Of course you can again solve this by just filling a temporary buffer in the lock and send the result back outside, like done in the FileMon demo.

A universal solution

What we have learned? Writing stable hook handlers is not only a matter of a good hooking library like EasyHook, but also a matter of experiences with windows API programming and issues that will show up in your own applications. Be careful when dealing with multi-thread code because the TDB won't fully protect you in this case. Don't call APIs or NET members which internally are using or waiting for other threads to complete. If you need to do this, use dedicated threads and exclude them from being intercepted. By the time you will write more and more stable hook handlers and don't worry about the very beginnings where things will cause the targets to crash or hang. With EasyHook you got a great utility in your hands, the rest depends on you!

Another side-effect that you should know about occurs when dealing with NET Remoting, again, is that you are usually implementing the shared class in the host application. This will cause the CLR to load your host assembly and initialize all associated static variables in related classes. This is why you shouldn't initialize any static variables in classes that are associated with ones used as remote interface.

2.12 A look into the future

With the latest version "EasyHook 2.5 Beta", all desired features are currently implemented. If you miss anything, don't hesitate to request a new feature. I plan a first release candidate during the next two months. Please report any bug you find, otherwise a stable release candidate is not possible!

Currently I am working on an OpenSource clone for the mentioned ProcessMonitor available at www.microsoft.com. Mine will only use user-mode hooks to accomplish its task and should be a good example of how to write hooking applications. I plan to publish it in late August 2008.

I don't have the time to test EasyHook all day long. I will always make sure that all shipped demos and test suites are working on all supported OS versions, but that is all I can do so far. Not only in case of OpenSource, is the customer the tester. But any bug you report will be fixed soon!