EasyHook

Table Of Content

1 Unmanaged API Reference

Please note that this reference only explains the unmanaged API. Refer to the documents “Introduction and Tutorial” and “Managed API Reference” for more information about how to write hooking handler and to understand the overall code logic.

1.1 Local Hooking API

In the following, methods marked with no asterix are available in user- AND kernel-mode, methods marked with one asterix are available in user-mode only and methods marked with two asterix are available in kernel-mode only. In general, if a method is available in both modes, it will behave the same and even share approx. 95% of the source code.

The following is an example of how to use the local hooking API in context of a driver; the user-mode version is very similar.

```
#include "EasyHook.h"


#define FORCE(expr) {if(!NT_SUCCESS(NtStatus = (expr))) goto ERROR_ABORT;}

static EASYHOOK_INTERFACE_API_v_1         Interface;

BOOLEAN KeCancelTimer_Hook(PKTIMER InTimer)
{
      PVOID                             Backup;
      PVOID                             CallStack[64];
      MODULE_INFORMATION                Mod;
      ULONG                             MethodCount;

      Interface.LhBarrierPointerToModule(0, 0);

      Interface.LhBarrierCallStackTrace(CallStack, 64, &MethodCount);

      Interface.LhBarrierGetCallingModule(&Mod);

      return KeCancelTimer(InTimer);
}


NTSTATUS RunTestSuite()
{
      HOOK_TRACE_INFO                   hHook = { NULL };
      NTSTATUS                          NtStatus;
      ULONG                             ACLEntries[1] = {0};
      UNICODE_STRING                    SymbolName;
      KTIMER                            Timer;
      BOOLEAN                        HasInterface = FALSE;
      PFILE_OBJECT                   hEasyHookDrv;

      FORCE(EasyHookQueryInterface(
            EASYHOOK_INTERFACE_v_1,
            &Interface,
            &hEasyHookDrv));

      HasInterface = TRUE;

      RtlInitUnicodeString(&SymbolName, L"KeCancelTimer");

    /*
        The following shows how to install and remove local hooks...
    */
    FORCE(Interface.LhInstallHook(
            MmGetSystemRoutineAddress(&SymbolName),
            KeCancelTimer_Hook,
            (PVOID)0x12345678,
            &hHook));

    // won't invoke the hook handle because hooks are inactive after
installation
    KeInitializeTimer(&Timer);

    KeCancelTimer(&Timer);

    // activate the hook for the current thread
    FORCE(Interface.LhSetInclusiveACL(ACLEntries, 1, &hHook));

    // will be redirected into the handler...
    KeCancelTimer(&Timer);

    // this won't unhook the entry point. But the associated handler is never
called again...
    Interface.LhUninstallHook(&hHook);

    // restores ALL entry points of currently pending removals issued by
LhUninstallHook()
    Interface.LhWaitForPendingRemovals();

      ObDereferenceObject(hEasyHookDrv);

      return STATUS_SUCCESS;

ERROR_ABORT:

      if(HasInterface)
      {
            ObDereferenceObject(hEasyHookDrv);

            KdPrint(("\n[Error]: \"%S\" (code: %d)\n",
                  Interface.RtlGetLastErrorString(),
                  Interface.RtlGetLastError()));
      }
      else
            KdPrint(("\n[Error]: \"Unable to obtain EasyHook interface.\"
(code: %d)\n", NtStatus));

    return NtStatus;
}
```

1.1.1 TRACED_HOOK_HANDLE

A traced handle was introduced, to make it easier or possible at all to update managed classes, containing a hook handle, if the native library is unloaded or someone removes such a hook or all hooks from unmanaged code. All in all it will make the thing much more stable. So what is a trace handle in particular?

A traced handle is pointer of the type:

```
typedef struct _HOOK_TRACE_INFO_
{
      PLOCAL_HOOK_INFO        Link;
}HOOK_TRACE_INFO, *TRACED_HOOK_HANDLE;
```

You will have to pre-allocate such a structure before you can create a hook with *LhInstallHook*. During hook installation, EasyHook will store an internal pointer in the above structure. You MUST NOT release this pre-allocated memory before either the native EasyHook library has been unloaded or you explicitly removed the hook with one of the hook removal routines.

If now somewhere your hook is removed, even if you don't know about it, the pointer in your handle can be set to *NULL*. The next time you use this zeroed handle, the corresponding method will fail with *STATUS_INVALID_PARAMETERXxx*. Without such a traced handle, EasyHook in contrast would dereference an invalid pointer and that usually is a bad thing especially in kernel mode.

1.1.2 LhInstallHook

Installs a hook at the given entry point, redirecting all calls to the given hooking method. The returned handle will either be released on library unloading or explicitly through *LhUninstallHook*() or *LhUninstallAllHooks*().

```
EASYHOOK_NT_EXPORT LhInstallHook(
            void* InEntryPoint,
            void * InHookProc,
            void * InCallback,
            TRACED_HOOK_HANDLE OutHandle);
```

Parameters

InEntryPoint

An entry point to hook. Not all entry points are hookable. In such a case *STATUS_NOT_SUPPORTED* will be returned.

InHookProc

The method that should be called instead of the given entry point. Please note that calling convention, parameter count and return value shall match EXACTLY!

InCallback

An uninterpreted callback later available through *LhBarrierGetCallback*().

OutHandle

The memory portion supplied by **OutHandle* is expected to be preallocated by the caller. This structure is then filled by the method on success and must stay valid for hook-life time. Only if you explicitly call one of the hook uninstallation APIs, you can safely release the handle memory.

Return values

STATUS_NO_MEMORY

Unable to allocate memory around the target entry point.

STATUS_NOT_SUPPORTED

The target entry point contains unsupported instructions.

STATUS_INSUFFICIENT_RESOURCES

The limit of *MAX_HOOK_COUNT* simultaneous hooks was reached.

1.1.3 LhUninstallHook

Removes the given hook. To also release associated resources, you will have to call *LhWaitForPendingRemovals*(). In any case your hook handler will never be executed again, after calling this method. Please note that this not mean that you handler is currently not executed anymore…

```
EASYHOOK_NT_EXPORT LhUninstallHook(TRACED_HOOK_HANDLE InHandle);
```

Parameters

InHandle
A traced hook handle. If the hook is already removed, this method will still return STATUS_SUCCESS.

1.1.4 *LhUninstallAllHooks

Removes ALL hooks (created with EasyHook) in the current process! To also release associated resources, you will have to call *LhWaitForPendingRemovals*().

```
EASYHOOK_NT_EXPORT LhUninstallAllHooks();
```

This method is not available in kernel to prevent concurrent drivers from removing each other's hooks.

1.1.5 LhWaitForPendingRemovals

For stability reasons, all resources associated with a hook have to be released if no thread is currently executing the handler. Separating this wait loop from the uninstallation method is a great performance gain, because you can release all hooks first (which is quite fast), and then wait for all removals simultaneously (which is then also quite fast).

```
EASYHOOK_NT_EXPORT LhWaitForPendingRemovals();
```

1.1.6 LhSetInclusiveACL

ATTENTION: In kernel-mode all ACL methods will always refer to processes instead of threads. The prototypes in the source code are changed appropriately, of course.

Sets an inclusive hook local ACL based on the given thread ID list. Global and local ACLs are always intersected. For example if the global ACL allows a set "*G*" of threads to be intercepted, and the local ACL allows a set "*L*" of threads to be intercepted, then the intersected set "$G \cap L$" will be intercepted. The "exclusive" and "inclusive" ACL types don't have any impact on the computation of the final set. Those are just helpers for you to construct a set of threads.

```
EASYHOOK_NT_EXPORT LhSetInclusiveACL(
            ULONG* InThreadIdList,
            ULONG InThreadCount,
            TRACED_HOOK_HANDLE InHandle);
```

Parameters

InThreadIdList

An array of thread IDs. If you specific zero for an entry in this array, it will be automatically replaced with the calling thread ID.

InThreadCount

The count of entries listed in the thread ID list. This value must not exceed MAX_ACE_COUNT!

InHandle

The hook handle whose local ACL is going to be set.

Return values

STATUS_INVALID_PARAMETER_2

The limit of *MAX_ACE_COUNT* ACL is violated by the given buffer.

1.1.7 LhSetExclusiveACL

ATTENTION: In kernel-mode all ACL methods will always refer to processes instead of threads. The prototypes in the source code are changed appropriately, of course.

Sets an exclusive hook local ACL based on the given thread ID list. Global and local ACLs are always intersected. For example if the global ACL allows a set "*G*" of threads to be intercepted, and the local ACL allows a set "*L*" of threads to be intercepted, then the set "$G \cap L$" will be intercepted. The "exclusive" and "inclusive" ACL types don't have any impact on the computation of the final set. Those are just helpers for you to construct a set of threads.

```
EASYHOOK_NT_EXPORT LhSetExclusiveACL(
            ULONG* InThreadIdList,
            ULONG InThreadCount,
            TRACED_HOOK_HANDLE InHandle);
```

Parameters

InThreadIdList

An array of thread IDs. If you specific zero for an entry in this array, it will be automatically replaced with the calling thread ID.

InThreadCount

The count of entries listed in the thread ID list. This value must not exceed MAX_ACE_COUNT!

InHandle

The hook handle whose local ACL is going to be set.

Return values

STATUS_INVALID_PARAMETER_2

The limit of *MAX_ACE_COUNT* ACL is violated by the given buffer.

1.1.8 LhSetGlobalInclusiveACL

ATTENTION: In kernel-mode all ACL methods will always refer to processes instead of threads. The prototypes in the source code are changed appropriately, of course.

Sets an inclusive global ACL based on the given thread ID list. Global and local ACLs are always intersected. For example if the global ACL allows a set "*G*" of threads to be intercepted, and the local ACL allows a set "*L*" of threads to be intercepted, then the set "*G* ∩ *L*" will be intercepted. The "exclusive" and "inclusive" ACL types don't have any impact on the computation of the final set. Those are just helpers for you to construct a set of threads.

```
EASYHOOK_NT_EXPORT LhSetGlobalExclusiveACL(
            ULONG* InThreadIdList,
            ULONG InThreadCount);
```

Parameters

InThreadIdList

An array of thread IDs. If you specific zero for an entry in this array, it will be automatically replaced with the calling thread ID.

InThreadCount

The count of entries listed in the thread ID list. This value must not exceed MAX_ACE_COUNT!

Return values

STATUS_INVALID_PARAMETER_2

The limit of *MAX_ACE_COUNT* ACL is violated by the given buffer.

1.1.9 LhSetGlobalExclusiveACL

ATTENTION: In kernel-mode all ACL methods will always refer to processes instead of threads. The prototypes in the source code are changed appropriately, of course.

Sets an exclusive global ACL based on the given thread ID list. Global and local ACLs are always intersected. For example if the global ACL allows a set "*G*" of threads to be intercepted, and the local ACL allows a set "*L*" of threads to be intercepted, then the set "*G* ∩ *L*" will be intercepted. The "exclusive" and "inclusive" ACL types don't have any impact on the computation of the final set. Those are just helpers for you to construct a set of threads.

```
EASYHOOK_NT_EXPORT LhSetGlobalExclusiveACL(
            ULONG* InThreadIdList,
            ULONG InThreadCount);
```

Parameters

InThreadIdList

An array of thread IDs. If you specific zero for an entry in this array, it will be automatically replaced with the calling thread ID.

InThreadCount

The count of entries listed in the thread ID list. This value must not exceed *MAX_ACE_COUNT*!

Return values

STATUS_INVALID_PARAMETER_2

The limit of *MAX_ACE_COUNT* ACL is violated by the given buffer.

1.1.10 *LhIsThreadIntercepted and **LhIsProcessIntercepted

This methods just check whether a given thread or process ID is intercepted by a given hook. The computation involves the intersection of the global and local ACL. You can use the method to check whether your ACL meets desired requirements.

```
EASYHOOK_NT_EXPORT LhIsThreadIntercepted(
            TRACED_HOOK_HANDLE InHook,
            ULONG InThreadID,
            BOOL* OutResult);

DRIVER_SHARED_API(NTSTATUS, LhIsProcessIntercepted(
            TRACED_HOOK_HANDLE InHook,
            ULONG InProcessID,
            BOOL* OutResult));
```

1.1.11 LhBarrierGetReturnAddress

Is expected to be called inside a hook handler. Otherwise it will fail with *STATUS_NOT_SUPPORTED*. The method retrieves the return address of the hook handler.

```
EASYHOOK_NT_EXPORT LhBarrierGetReturnAddress(PVOID* OutValue);
```

1.1.12 LhBarrierGetAddressOfReturnAddress

Is expected to be called inside a hook handler. Otherwise it will fail with *STATUS_NOT_SUPPORTED*. The method retrieves the address of the return address of the hook handler.

```
EASYHOOK_NT_EXPORT LhBarrierGetAddressOfReturnAddress(PVOID** OutValue);
```

1.1.13 LhBarrierGetCallback

Is expected to be called inside a hook handler. Otherwise it will fail with *STATUS_NOT_SUPPORTED*. The method retrieves the callback initially passed to the related *LhInstallHook* call.

```
EASYHOOK_NT_EXPORT LhBarrierGetCallback (PVOID** OutValue);
```

1.1.14 LhBarrierBeginStackTrace

Is expected to be called inside a hook handler. Otherwise it will fail with *STATUS_NOT_SUPPORTED*. The method temporarily restores the call stack to allow stack traces. You have to pass the stored backup pointer to *LhBarrierEndStackTrace* BEFORE leaving the handler, otherwise the application will be left in an undefined state!

```
EASYHOOK_NT_EXPORT LhBarrierBeginStackTrace(PVOID* OutBackup)
```

1.1.15 LhBarrierEndStackTrace

Is expected to be called inside a hook handler. Otherwise it will fail with *STATUS_NOT_SUPPORTED*. You have to pass the backup pointer obtained with *LhBarrierBeginStackTrace*().

```
EASYHOOK_NT_EXPORT LhBarrierEndStackTrace(PVOID InBackup);
```

1.1.16 MODULE_INFORMATION

The methods *LhBarrierPointerToModule* and *LhBarrierGetCallingModule* will use the following type to expose module information:

```
typedef struct _MODULE_INFORMATION_* PMODULE_INFORMATION;

typedef struct _MODULE_INFORMATION_
{
      PMODULE_INFORMATION           Next;
      UCHAR*                        BaseAddress;
      ULONG                         ImageSize;
      CHAR                          Path[256];
      PCHAR                         ModuleName;
}MODULE_INFORMATION;
```

The linked list assembled by *Next* is always set to *NULL* by the time module information leaves the EasyHook core. The fields should be self-explaining this is why I don't cover them here.

1.1.17 LhEnumerateModules

You should enumerate modules in two calls. The first one shall have *OutModules* set to *NULL*, *InMaxModuleCount* to zero and will receive *OutModuleCount*. Now you can allocate a buffer large enough to hold all modules and pass it in a second call. If your buffer is too small, STATUS_BUFFER_TOO_SMALL is returned but all available entries are still filled…

```
EASYHOOK_NT_EXPORT LhEnumModules(
            HMODULE* OutModules,
            ULONG InMaxModuleCount,
            ULONG* OutModuleCount);
```

Note that in kernel-mode this method enumerates kernel mode drivers whereas in user-mode loaded libraries. For performance reasons only module pointers are returned. You may use *LhBarrierPointerToModule* for each entry to get the full module information.

1.1.18 LhBarrierPointerToModule

Translates the given pointer (likely a method) to its owning module if possible. If no matching module could be found, *STATUS_NOT_FOUND* is returned.

```
EASYHOOK_NT_EXPORT LhBarrierPointerToModule(
              PVOID InPointer,
              MODULE_INFORMATION* OutModule);
```

1.1.19 LhBarrierGetCallingModule

Saves the calling module information in the given buffer; please note that only unmanaged modules are recognized. If the calling module cannot be resolved, *STATUS_NOT_FOUND* is returned.

```
EASYHOOK_NT_EXPORT LhBarrierGetCallingModule(MODULE_INFORMATION* OutModule);
```

1.1.20 LhBarrierCallStackTrace

Creates a call stack trace. This is only available since Windows XP. Only method pointers are returned, because in kernel mode there is not enough stack space available to statically query all modules at once. To enumerate modules on the stack, just loop through the method array and use *LhBarrierPointerToModule* for each entry.

```
EASYHOOK_NT_EXPORT LhBarrierCallStackTrace(
            PVOID* OutMethodArray,
            ULONG InMaxMethodCount,
            ULONG* OutMethodCount);
```

Parameters

OutMethodArray

Receives all method entries on the call stack. Cannot be set to *NULL*!

InMaxMethodCount

The maximum count of method pointers the given buffer can hold. At maximum 64 pointers are captured…

OutMethodCount

Receives the actual count of method entries on the call stack.

1.1.21 RtlGetLastError

Returns the last error code; this is either *GetLastError* directly after the error occurred or the returned *NTSTATUS* depending on the API.

```
EASYHOOK_NT_EXPORT RtlGetLastError();
```

1.1.22 RtlGetLastErrorString

Returns advanced error information as string. I always recommend including this in any error report, because it is the only way to determine the exact point of failure in the source code.

```
PWCHAR RtlGetLastErrorString();
```

1.2 Driver related API

Supporting kernel mode hooks has never been that easy. In the following chapter I will explain the three core APIs that are used to install all required drivers and obtain the EasyHook API in kernel mode.

Please note that to use the "EasyHook.h" header in a driver you will have to define a macro named "DRIVER", for example in the preprocessor settings.

1.2.1 RhInstallSupportDriver

Installs the EasyHook support driver. This will allow your driver to successfully obtain the EasyHook driver API using *EasyHookQueryInterface*. Once installed, the driver will be immediately marked for deletion and so be vanished on next system start or driver shutdown.

```
EASYHOOK_NT_EXPORT RhInstallSupportDriver();
```

1.2.2 RhInstallDriver

Simplifies the installation of 3th party drivers intended for kernel mode hooking. It will immediately mark the driver for deletion, so that it will be vanished after shutdown or next system start. *InDriverPath* specifies a relative or full path to the driver's executable. *InDriverName* is a name to register the driver in the service control manager.

```
EASYHOOK_NT_EXPORT RhInstallDriver(
            WCHAR* InDriverPath,
            WCHAR* InDriverName);
```

1.2.3 **EasyHookQueryInterface

To get access to the EasyHook support driver, you may call the following method in a driver context.
Please keep in mind that this will only work if EasyHook was loaded into the kernel!
If you don't need the interface anymore, you have to release the file object with *ObDereferenceObject*.

```
static NTSTATUS EasyHookQueryInterface(
            ULONG InInterfaceVersion,
            PVOID OutInterface,
            PFILE_OBJECT* OutEasyHookDrv);
```

Parameters

InInterfaceVersion

The desired interface version. Any future EasyHook driver will ALWAYS be backward compatible. This is the reason why I provide a flexible interface mechanism.

OutInterface

A pointer to the interface structure being filled with data. If you specify *EASYHOOK_INTERFACE_v_1* as InInterfaceVersion, you will have to provide a pointer to a *EASYHOOK_INTERFACE_API_v_1* structure, for example...

OutEasyHookDrv

A reference to the EasyHook driver. Make sure that you dereference it if you don't need the interface any longer! As long as you keep this handle, the EasyHook driver CAN'T be unloaded...

1.2.4 EASYHOOK_INTERFACE_API_v_1

Currently this is the only supported interface that may be queried with

```
EASYHOOK_INTERFACE_API_v_1          Interface;
PFILE_OBJECT                        hEasyHookDrv = NULL;

EasyHookQueryInterface(EASYHOOK_INTERFACE_v_1, &Interface, &hEasyHookDrv);

// TODO: work with the interface…
Interface.LhInstallHook(...);

ObDereferenceObject(hEasyHookDrv);
```

, and exports all of the mentioned kernel methods in the local hooking API paragraph.

1.2.5 Preparing the build environment

As first step you should download the latest Windows Driver Kit from https://connect.microsoft.com/ and install it to "C:\WinDDK" for example. You have to register and sign in to Microsoft Passport I believe but after all it is still free.

Then you have to add a system wide environment variable called "WDKROOT" with the value of your installation path.

From now on I assume that you extracted the EasyHook source code archive to "C:\EasyHook"; so that the project solution is located in "C:\EasyHook\EasyHook.sln".

To install a driver on Windows Vista, you have to sign it. Now open a root-shell and type the following commands:

```
> Bcdedit –set TESTSIGNING ON
```

You need to restart the PC. After this we are ready to create a test certificate:

```
> cd c:\winddk\bin\SelfSign

> MakeCert -r -pe -ss EasyHookTestCertStore -n  "CN=EasyHookTestCert"
"c:\EasyHook\EasyHookTestCert.cer"
```

Now open the certificate in the Explorer by double clicking it. Currently the certificate is an untrusted one. To make Windows trusting the certificate, click "Install certificate" – "Next" – "place all certificates in the following store" – "Trusted Root Certification Authorities" – "OK" – "Next" – "Finish" and do the same procedure to add it to the "Trusted Publishers" store.

The project comes with a post-build event that relies on the fact that a certificate named "*EasyHookTestCert*" is in the "*EasyHookTestCertStore* ". This will automatically sign the drivers after each build and you have to care about nothing.

Now we are done and you can start building the project!

1.2.6 Service management

By default *RhInstallSupportDriver* checks whether the EasyHook driver is already installed. If this is the case, nothing is done. If not, then it just installs the driver. The driver is always deleted before the method returns. If you stop it or restart the system, it will be vanished. If any kernel hook is currently applied, then normally the driver can't be unloaded. In the worst case all hooks are removed, so a BSOD shouldn't occur in any case!

But if you want to frequently recompile it, you need to make sure that the driver has been unloaded and removed from the service control manager, before you restart the application. Otherwise your newly compiled driver won't be loaded because the old one is already there…

If you want to remove the driver from the service manager, just type "sc stop EasyHook32Drv.sys" into a root shell. On a 64-Bit platform you would need to replace the "32" with "64".

1.3 Debugging API

While writing hook handlers you might encounter situations in which you only get handles, but want to know which thread ID, process ID or file name is hidden behind. No debugging API is available in kernel-mode!

The following code snipped shows how to use the debugging API:

```
HANDLE          Handle = CreateEventA(NULL, TRUE, FALSE, "MyEvent");
ULONG           RequiredSize;
ULONG           RealThreadId;
ULONG           ThreadId;
UNICODE_STRING* NameBuffer = NULL;

// handle to name
if(!SUCCEEDED(NtStatus = DbgHandleToObjectName(
        Handle,
        NULL, 0,
        &RequiredSize)))
    goto ERROR_ABORT;

NameBuffer = (UNICODE_STRING*)malloc(RequiredSize);

FORCE(DbgHandleToObjectName(
        Handle,
        NameBuffer,
        RequiredSize,
        &RequiredSize));

printf("\n[Info]: Event name is \"%S\".\n", NameBuffer->Buffer);

// handle to thread ID
Handle = CreateThread(
        NULL, 0, NULL, NULL,
        CREATE_SUSPENDED,
        &RealThreadId);

FORCE(DbgGetThreadIdByHandle(Handle, &ThreadId));

if(ThreadId != RealThreadId)
    throw;
```

Those methods above don’t use a debugger in its original sense and thus are fast enough for most purposes.

1.3.1 DbgAttachDebugger

There is only one method really requiring an attached debugger. It is not exported but you will have to attach a debugger explicitly to let EasyHook use it. Internally this will allow EasyHook to relocate RIP relative addressing, which may only occur on 64-Bit. In general the chance for RIP relative addressing in the first five entry point bytes is quite small. Therefore I decided to disable the detection by default and instead let EasyHook take advantage of the resulting performance gain by skipping the disassembling step. You should only enable debugging with this method, if hooking fails without it or you know for sure that there will be RIP relative addressing in the first five bytes (this does not apply to any common windows API I know).

```
EASYHOOK_NT_EXPORT DbgAttachDebugger();
```

By design there is no real world case in which this method should fail, and if it does there is no way to do something against. I recommend just calling it without checking the result.

1.3.2 DbgGetThreadIdByHandle

Translates the given thread handle back to its thread ID if possible. *InHandle* shall be a thread handle with *THREAD_QUERY_INFORMATION* access.

```
EASYHOOK_NT_EXPORT DbgGetThreadIdByHandle(
                HANDLE InThreadHandle,
                ULONG* OutThreadId);
```

1.3.3 DbgGetProcessIdByHandle

Translates the given thread handle back to its thread ID if possible. *InHandle* shall be a thread handle with *PROCESS_QUERY_INFORMATION* access.

```
EASYHOOK_NT_EXPORT DbgGetProcessIdByHandle(
                HANDLE InProcessHandle,
                ULONG* OutProcessId);
```

1.3.4 DbgHandleToObjectName

Translates the given handle back to its object name if possible. *InHandle* shall refer to a named object. You should call this method twice. Firstly to query the required buffer size and secondly to actually query the object name with the allocated buffer.

```
EASYHOOK_NT_EXPORT DbgHandleToObjectName(
                HANDLE InNamedHandle,
                PUNICODE_STRING OutNameBuffer,
                ULONG InBufferSize,
                ULONG* OutRequiredSize);
```

Parameters

InNamedHandle

A valid file, event, section, etc.

OutNameBuffer

A buffer large enough to hold the kernel space object name. To query the required size in bytes, set this parameter to *NULL*.

InBufferSize

The maximum size in bytes the given buffer can hold.

OutRequiredSize

Receives the required size in bytes. This parameter can be *NULL*.

1.4 Remote hooking API

This chapter covers library injection. With unmanaged injection you have several problems left. And this is why I recommend you to use managed injection whenever possible. For example, no system service is used and also no WOW64 bypass. This way it is not possible to hook into other terminal sessions or through WOW64 boundaries using the unmanaged API.

1.4.1 RhCreateStealthRemoteThread

Using Remote thread creation on bad purpose is not new. This is why many AV software (like Windows Defender) will mark processes that do so as SUSPECT. To workaround this issue I introduce a new stealth thread creation. This is done by hijacking an existing thread and letting it execute an invokation stub which itself creates a thread using *CreateThread* in the target process. This thread again will execute the given remote thread proc. After such a stealth remote thread has been created, the hijacked thread will continue execution just like nothing has happened.

The problem with this approach is, that there are several bad circumstances in which the target process might be damaged or crashed. This also won't work if no running thread is available in the target process (for example processes created with *RhCreateAndInject*). But I thought it would be something new to play with ;-). Any sane process shouldn't be damaged by this approach but there is no guarantee.

Unfortunately, this approach does not work for managed injections. *ICLRHost::ExecuteInDefaultAppDomain* will always fail.

```
EASYHOOK_NT_EXPORT RhCreateStealthRemoteThread(
            ULONG InTargetPID,
            LPTHREAD_START_ROUTINE InRemoteRoutine,
            PVOID InRemoteParam,
            HANDLE* OutRemoteThread);
```

Parameters

InTargetPID

The process ID of the target.

InRemoteRoutine

A usual thread start routine in the target process space.

InRemoteParam

An uninterpreted value passed to the remote start routine.

OutRemoteThread

Receives a handle to the remote thread which is valid in the current process.

Returns

STATUS_WOW_ASSERTION

Remote thread creation through WOW64 boundaries is not supported.

STATUS_NOT_FOUND

The target process could not be found.

STATUS_ACCESS_DENIED

The target process could not be accessed properly.

STATUS_NOT_SUPPORTED

There is no running thread in the target.

1.4.2 RhInjectLibrary

Injects a library into the target process. This is a very stable operation. The problem so far is, that only the NET layer will support injection through WOW64 boundaries and into other terminal sessions. It is quite complex to realize with unmanaged code and that's why it is not supported!

If you really need this feature I highly recommend to at least look at C++.NET because using the managed injection can speed up your development progress about orders of magnitudes. I know by experience that writing the required multi-process injection code in any unmanaged language is a rather daunting task!

```
EASYHOOK_NT_EXPORT RhInjectLibrary(
            ULONG InTargetPID,
            ULONG InWakeUpTID,
            ULONG InInjectionOptions,
            WCHAR* InLibraryPath_x86,
            WCHAR* InLibraryPath_x64,
            PVOID InPassThruBuffer,
            ULONG InPassThruSize);
```

InTargetPID

The process in which the library should be injected.

InWakeUpTID

This parameter is reserved for internal use only and shall be set to zero.

InInjectionOptions

Managed and stealth injection cannot be combined!

EASYHOOK_INJECT_DEFAULT

No special behavior. The given libraries are expected to be unmanaged DLLs. Further they should export an entry point named "NativeInjectionEntryPoint" (in case of 64-bit) and "_NativeInjectionEntryPoint@4" (in case of 32-bit). The expected entry point signature is *REMOTE_ENTRY_POINT*.

EASYHOOK_INJECT_MANAGED

The given user library is a NET assembly. Further they should export a class named "EasyHook.InjectionLoader" with a static method named "Main". The signature of this method is expected to be "int (String)". Please refer to the managed injection loader of EasyHook for more information about writing such managed entry points.

EASYHOOK_INJECT_STEALTH

Uses the experimental stealth thread creation. If it fails you may try it with default settings.

InLibraryPath_x86

A relative or absolute path to the 32-bit version of the user library being injected. If you don't want to inject into 32-Bit processes, you may set this parameter to *NULL*.

InLibraryPath_x64

A relative or absolute path to the 64-bit version of the user library being injected. If you don't want to inject into 64-Bit processes, you may set this parameter to *NULL*.

InPassThruBuffer

An optional buffer containg data to be passed to the injection entry point. Such data is available in both, the managed and unmanaged user library entry points. Set to *NULL* if no used.

InPassThruSize

Specifies the size in bytes of the pass thru data. If *InPassThruBuffer* is *NULL*, this parameter shall also be zero.

Returns

STATUS_INVALID_PARAMETER_5

The given 64-Bit library could not be found, does not export the expected entry point or could not be loaded into the target process, probably due to missing dependencies.

STATUS_INVALID_PARAMETER_4

The given 32-Bit library could not be found, does not export the expected entry point or could not be loaded into the target process, probably due to missing dependencies.

STATUS_WOW_ASSERTION

Injection through WOW64 boundaries is not supported.

STATUS_NOT_FOUND

The target process could not be found.

STATUS_ACCESS_DENIED

The target process could not be accessed properly or remote thread creation failed.

1.4.3 REMOTE_ENTRY_INFO

Your unmanaged and managed entry point will be called with the following structure after injection:

```
typedef struct _REMOTE_ENTRY_INFO_
{
      ULONG              HostPID;
      UCHAR*             UserData;
      ULONG              UserDataSize;
}REMOTE_ENTRY_INFO;
```

The fields should be self explaining. Please note that this structure is void after you return from the entry point!

1.4.4 RhCreateAndInject

Creates a suspended process and immediately injects the user library. This is done BEFORE any of the usual process initialization is called. When the injection is made, NO thread has actually executed any instruction so far... It is just like your library entry point is the first thing executed in such a process and you can allow the original execution to take place by calling *RhWakeUpProcess* in the injected library. But even that is no requirement for the process to work...

```
EASYHOOK_NT_EXPORT RhCreateAndInject(
            WCHAR* InEXEPath,
            WCHAR* InCommandLine,
            ULONG InInjectionOptions,
            WCHAR* InLibraryPath_x86,
            WCHAR* InLibraryPath_x64,
            PVOID InPassThruBuffer,
            ULONG InPassThruSize,
            ULONG* OutProcessId);
```

InEXEPath

A relative or absolute path to the EXE file of the process being created.

InCommandLine

Optional command line parameters for process creation.

All other parameters and return values are equal to those of *RhInjectLibrary*.

1.4.5 RhIsX64System

Especially when it comes to drivers, this method will help you to determine the underlying OS bitness. Note that a process running under WOW64 cannot install a 32-bit driver…

```
EASYHOOK_BOOL_EXPORT RhIsX64System();
```

It returns TRUE if a 64-bit OS version is installed and FALSE otherwise.

1.4.6 RhIsX64Process

Detects whether a given process is a 64-bit binary or not. Please note that the caller needs *PROCESS_QUERY_INFORMATION* to the target.

```
EASYHOOK_NT_EXPORT RhIsX64Process(
            ULONG InProcessId,
            BOOL* OutResult);
```

1.4.7 RhIsAdministrator

Just checks whether the caller is able to install a system service which usually is equal to be running with administrator privileges.

```
EASYHOOK_BOOL_EXPORT RhIsAdministrator();
```

Returns TRUE if the caller has admin privileges, FALSE otherwise.

1.4.8 RhWakeUpProcess

Used in conjunction with RhCreateAndInject and wakes up the injection target. You should call this method in the injection library entry point, after all hooks (or whatever) are applied.

```
EASYHOOK_NT_EXPORT RhWakeUpProcess();
```