

Hooking COM Objects: Intercepting Calls to COM Interfaces

apriorit.com/dev-blog/222-intercepting-com-calls



Wed, 03 December 2014 10:12

1

The current article was written to help you get familiar with the procedure of implementing COM interface hooking. Here you'll find: theory, functional code samples, and clear explanations. If you are acquainted with user-mode API hooks, it will be apparent to you that the process of API hooking shares some features with hooking COM objects (not only in methods used, but also in their purpose). However COM technology has its own specifics, so the two processes differ in some aspects. The process of COM object hooking has two major approaches. Each approach will be considered in details; we will consider the upsides and downsides of each of them. To make the process described here easier to follow, we have tried to make the code examples given here simple in order to focus on what is relevant for our task.

More information about the API hooking process is available in the [Windows API Hooking](#) post.

Contents:

1. Learning basics of COM objects
 2. Practice
- Using proxy object
- 2.2. Modifying Vtable
 3. Summary
 4. References

1. Learning basics of COM objects

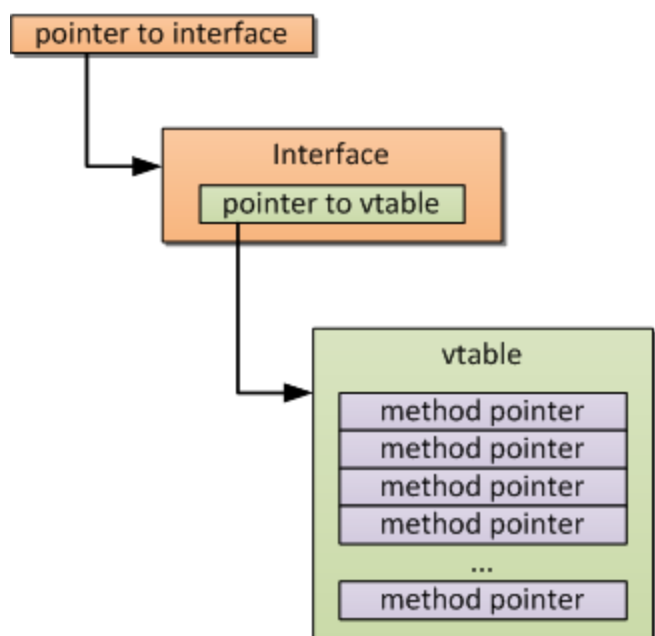
In general, to hook COM interface methods we need to intercept calls to COM objects, but before we commence to this task, it would be useful to discuss some basic principles of the COM technology. This section will introduce you to the basics, so if this is what you already know, you can jump to the practical section, as this introductory section will have nothing new to teach you.

COM classes are designed to support the implementation of several interfaces. These interfaces have a common limitation: they must originate from IUnknown. Its functions are to count references and to get pointers to interfaces, which other objects implement. An interface can be uniquely identified by its IID (an interface ID, which is globally unique). Interface pointers are used by clients to perform calls for methods of a COM object.

Thanks to this, COM objects gain independence on the binary level. This saves us from the process of recompiling clients in case the corresponding COM server changes, but the server must still provide the same interfaces after it was modified. Not only that, there is also room for changes, which allows implementing a custom replacement for the server.

The virtual method table (also referred to as vtable) is basically a pointer array, which performs calls of COM interface methods. Every pointer is a pointer to class methods, taking into account the order of declaration. Every COM class starts with a pointer to vtable with no exceptions. So to invoke any method, a client must use the corresponding pointer to perform a call.

The context (the client process context or that of any other process) in which a COM server works is also important. In the client process context, the server is represented by a DLL, which loads into the client process. In any other case the server is run



as another process (it may work either on a local or a remote machine). The communications between a client and the server are performed with the help of a special Proxy/Stub DLL, which redirects client calls to the server.

To make a COM server easy to access, you need to register it in the system registry. If you search different sources on the Internet, you will find some functions that serve as actors in creation of COM instances; the most common examples are `CoGetClassObject` , `CoCreateInstanceEx` , and `CoCreateInstance` . The `CoCreateInstance` is a preferable choice in most cases.

For more information, refer to [MSDN](#). Also check links at the end of the article.

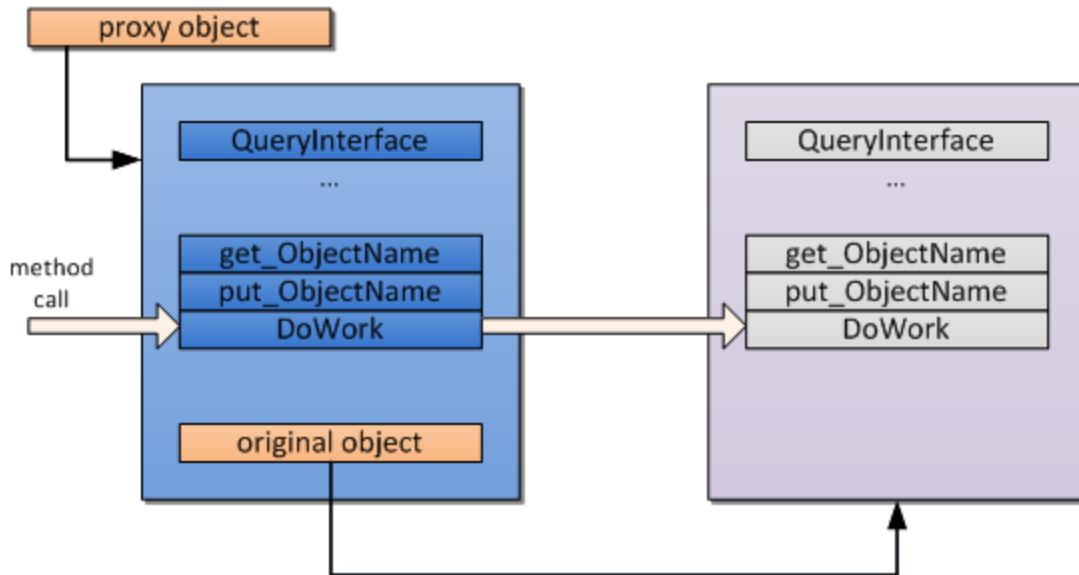
2. Practical example

Here we will discuss interception of calls to a COM interface. There are a number of methods we can use. First of all, we can make registry modifications. We can also use `CoTreatAsClass` or `CoGetInterceptor` function. The two approaches we are going to consider are: using proxy objects and virtual method table modification. Both approaches will be considered in details mentioning their upsides and downsides so that you can decide for yourself which one works best for you.

We have provided a fragment of code that illustrates how you can implement COM object in C++ on the example of a simplest COM server DLL and corresponding client applications. The code we used for hooking COM objects is also cited here. These examples will be enough to help you understand the described approaches.

To start with, we start a client application without installing hooks. Invoke the `regsvr32 ComSample.dll` command to register the COM server. Then start `ComSampleClient.exe` or `SrciptCleint.js`, which allows us to analyze the work of a client. After that we can finally commence to setting hooks.

2.1 Approach #1: Proxy object



COM is based around binary encapsulation. A client can use an interface to establish communication with any COM server. In the same time, you can modify the server however you want without the need to recompile clients, which spares much time and energy if you need something to be changed after the clients are compiled and working. So this is the feature we will utilize to perform the interception of calls to the COM server.

Using this technique, we replace a created instance of an object, which is contained in an intercepted request for the creation of a COM object, with a custom proxy COM object, which has an interface identical to the source object. This gives the client code an ability to interact with it as if it was the source object. In a common case, there is a pointer to the source object within a proxy object, which enables it to call methods of original objects.

So there is a proxy object and target object interfaces implemented by the proxy. We have selected only the **ISampleObject** interface for demonstration. The ATL implementation of the proxy class looks as follows:

```

class ATL_NO_VTABLE CSampleObjectProxy :
    public ATL::CComObjectRootEx<ATL::CComMultiThreadModel>,
    public ATL::CComCoClass<CSampleObjectProxy,
&CLSID_SampleObject>,
    public ATL::IDispatchImpl<ISampleObject, &IID_ISampleObject,
&LIBID_ComSampleLib, 1, 0>
{
public:
    CSampleObjectProxy();
    DECLARE_NO_REGISTRY()
    BEGIN_COM_MAP(CSampleObjectProxy)
        COM_INTERFACE_ENTRY(ISampleObject)
        COM_INTERFACE_ENTRY(IDispatch)
    END_COM_MAP()
    DECLARE_PROTECT_FINAL_CONSTRUCT()
public:
    HRESULT FinalConstruct();
    void FinalRelease();
public:
    HRESULT static CreateInstance(IUnknown* original, REFIID riid,
void **ppvObject);
public:
    STDMETHOD(get_ObjectName)(BSTR* pVal);
    STDMETHOD(put_ObjectName)(BSTR newVal);
    STDMETHOD(DoWork)(LONG arg1, LONG arg2, LONG* result);
    ...

```

```

};

STDMETHODIMP CSampleObjectProxy::get_ObjectName(BSTR* pVal)
{
    return m_Name.CopyTo(pVal);
}

STDMETHODIMP CSampleObjectProxy::DoWork(LONG arg1, LONG arg2, LONG*
result)
{
    *result = 42;
    return S_OK;
}

STDMETHODIMP CSampleObjectProxy::put_ObjectName(BSTR newVal)
{
    return m_OriginalObject->put_ObjectName(newVal);
}

```

When implementing the proxy object, there may be methods you don't need, but nevertheless these methods still have to be implemented. Unfortunately, this is a downside, which we cannot overcome.

For the next step, we perform the interception of a call for creation of an object file we are interested in and substitute it using our proxy. For this purpose, we use Windows API functions. The most commonly used one is `CoCreateInstance` that can create COM objects.

For training purposes of intercepting the object creation, you can use the `mhook` library for hooking the `CoCreateInstance` and `CoGetClassObject`. If you want, it won't be difficult to find information on this API hooking technique on the Internet. There is also some information you may find useful in the [Easy way to set up global API hooks](#) article by Sergey Podobry on our site.

The `CoCreateInstance` function is implemented as follows:

```
HRESULT WINAPI Hook::CoCreateInstance(REFCLSID rclsid, LPUNKNOWN
pUnkOuter, DWORD dwClsContext, REFIID riid, LPVOID* ppv)
{
    if (rclsid == CLSID_SampleObject)
    {
        if (pUnkOuter)
            return CLASS_E_NOAGGREGATION;

        ATL::CComPtr<IUnknown> originalObject;

        HRESULT hr = Original::CoCreateInstance(rclsid, pUnkOuter,
dwClsContext, riid, (void**)&originalObject);

        if (FAILED(hr))
            return hr;

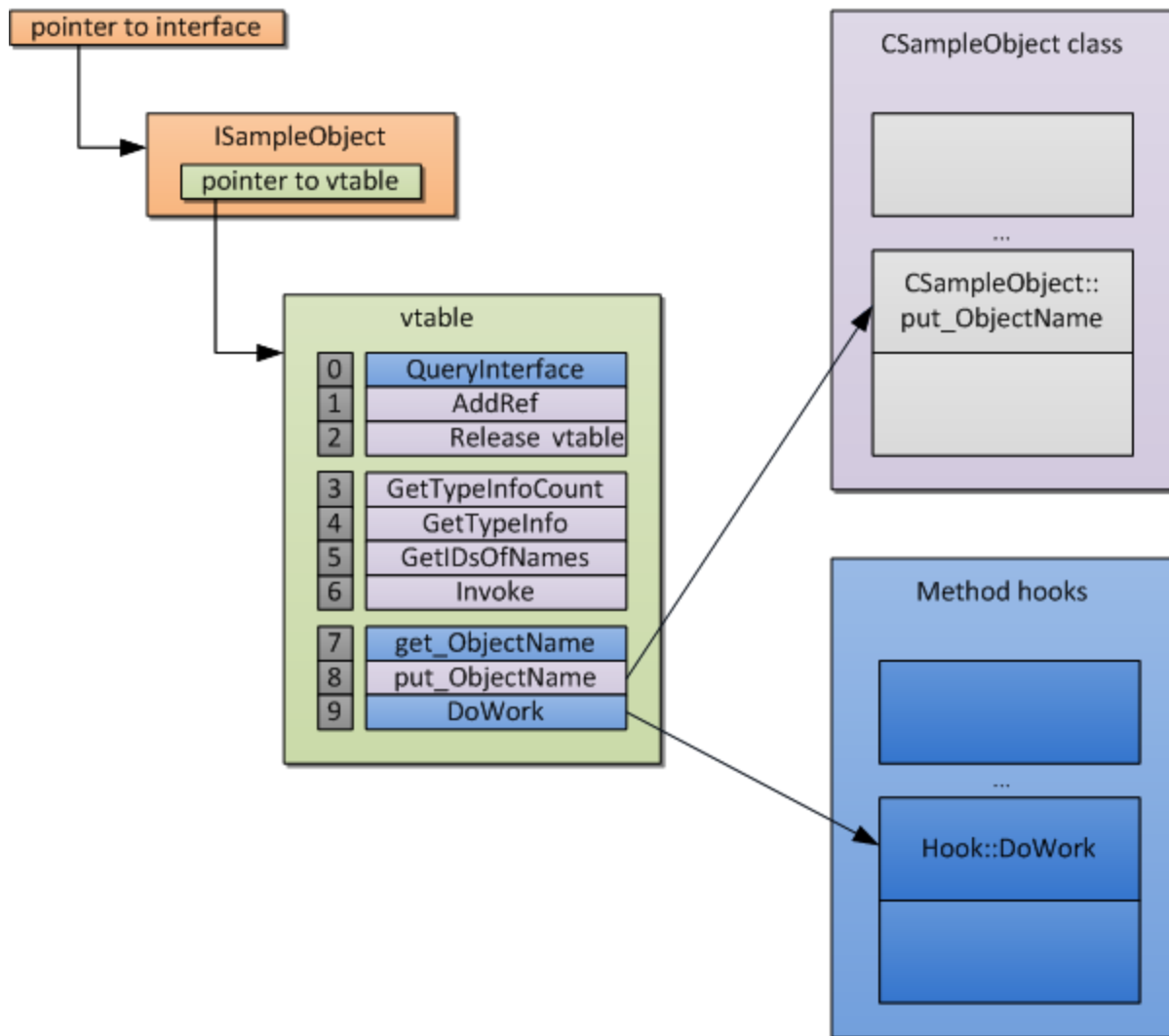
        return CSampleObjectProxy::CreateInstance(originalObject,
riid, ppv);
    }

    return Original::CoCreateInstance(rclsid, pUnkOuter,
dwClsContext, riid, ppv);
}
```

If you want to examine how a proxy object functions, you need to add the full name of `ComInterceptObj.dll` to the `AppInit_DLLs` registry value (the value can be found in `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows`). To test if the interception of the calls to an object works, simply start `ComSampleClient.exe` or `ScriptClient.js` and see the results.

2.2 Approach #2: Vtable patching

We can likewise hook COM object calls by making modifications to the virtual method table (**vtable**) of an object. As we have mentioned earlier, it is a container for pointers to public methods of a COM object. We can easily substitute them with the pointers to hook functions.



Talking about the advantages of this method, the difference from the first approach is that here we don't need to set hooks before the pointers to a target object have been acquired by a client. If there is access to a pointer to an object, you can set a hook there.

The following code from the **HookMethod** function illustrates the process of setting a hook for a COM interface method:

```

HRESULT HookMethod(IUnknown* original, PVOID proxyMethod, PVOID*
originalMethod, DWORD vtableOffset)
{
    PVOID* originalVtable = *(PVOID**)original;
    if (originalVtable[vtableOffset] == proxyMethod)
        return S_OK;
    *originalMethod = originalVtable[vtableOffset];
    originalVtable[vtableOffset] = proxyMethod;
    return S_OK;
}

```

Then we use the `InstallComInterfaceHooks` function, which allows us to hook the interface methods belonging to `ISampleObject` :

```

HRESULT InstallComInterfaceHooks(IUnknown* originalInterface)
{
    // Only single instance of a target object is supported in the
    sample

    if (g_Context.get())

        return E_FAIL;

    ATL::CComPtr<ISampleObject> so;

    HRESULT hr = originalInterface-
>QueryInterface(IID_ISampleObject, (void**)&so);

    if (FAILED(hr))

        return hr; // we need this interface to be present

    // remove protection from the vtable

    DWORD dwOld = 0;

    if(!::VirtualProtect(*(PVOID**)(originalInterface),
sizeof(LONG_PTR), PAGE_EXECUTE_READWRITE, &dwOld))

        return E_FAIL;

    // hook interface methods

    g_Context.reset(new Context);

    HookMethod(so, (PVOID)Hook::QueryInterface, &g_Context-
>m_OriginalQueryInterface, 0);

    HookMethod(so, (PVOID)Hook::get_ObjectName, &g_Context-
>m_OriginalGetObjectName, 7);

    HookMethod(so, (PVOID)Hook::DoWork, &g_Context-
>m_OriginalDoWork, 9);

    return S_OK;
}

```

Before you try to work with vtable, you need to make sure it can be modified: sometimes it may be located in a read-only area, which makes it write-protected. If it is, use `VirtualProtect`, which will remove the protection.

The target object related data is stored in the `g_Context` variable. This variable is a structure by itself. For demonstration purposes, we've made the `g_Context` variable global. Please note that it allows only one target object to exist at a time.

The code of our hook function looks as follows:

```

typedef HRESULT (WINAPI *QueryInterface_T)(IUnknown* This, REFIID riid, void **ppvObject);

STDMETHODIMP Hook::QueryInterface(IUnknown* This, REFIID riid, void **ppvObject)
{
    QueryInterface_T qi = (QueryInterface_T)g_Context->m_OriginalQueryInterface;

    HRESULT hr = qi(This, riid, ppvObject);

    return hr;
}

STDMETHODIMP Hook::get_ObjectName(IUnknown* This, BSTR* pVal)
{
    return g_Context->m_Name.CopyTo(pVal);
}

STDMETHODIMP Hook::DoWork(IUnknown* This, LONG arg1, LONG arg2, LONG* result)
{
    *result = 42;

    return S_OK;
}

```

Now for the definitions of the hook functions. Their prototypes and the prototypes of the target interface methods are identical. The difference here is that the former are not class methods, but free functions with an additional parameter – the this pointer. This is easily explained: we declare COM methods as `stdcall`, while this serves as an implicit stack parameter.

So if you decide to stick to this approach, you should keep a few things in mind. The first thing you must remember is that when setting a hook, you affect all objects that have the same class, not just the COM object instance you work with. However this does not mean that if some classes implement the hooked interface, all of them will be affected. In this case, if all interface instances must be intercepted, you need to modify the vtable for each of the corresponding classes.

There also may be cases when you need an object specific data to be stored. For this purpose, you need to have a static memory area, in which a context collection accessible by the value of a target object pointer will be stored. The static collection also needs synchronization in case you need to have multithreaded access to be available for the target object. And of course keep track of lifetime of a target object.

Please also bear in mind that calling a target method, which has been hooked, by an interface pointer may have unwanted side effects. This will simply result in an access to vtable and, as a result, the hook function will be called. This is certainly one thing you want to avoid. So to prevent this from happening, the pointer to the source method must be available and must call the method directly.

One more thing about this approach: if a method has been hooked, it must not be hooked the second time. If you allows this to happen, the second attempt to hook a method will overwrite any pointer to a source method you have previously saved. However obvious it can be, it must be mentioned, because this is something you doesn't pay much attention to while using this method, and, as a result, it is the most frequently overlooked point.

There are also other advantages to this approach: if there are methods that do not require interception, there is no need to implement hooks for them. There is also no need to perform interception of object creation. So a lot of extra work can be avoided using this approach.

Now, to see this sample in action, simply open the `AppInit_DLLs` registry value and enter the full name of the `ComInterceptVtablePatch.dll` dll in it. We have already performed similar action earlier. After that, run the client.

3. Summary

We have considered two approaches from which you can choose when hooking COM objects. Both have their own pros and cons.

The implementation of the proxy object approach is much easier (this becomes even clearer when you attempt to implement sophisticated logic in it). The difficulty here is that the replacement of a target object has to take place before a pointer to the source object has been obtained by a client. This task may turn out to be impossible or at least very difficult in certain situations. In addition, both the proxy and the target object must have identical interfaces regardless of the number of methods for interception. Be aware that we have

shown only simple examples here, while in real world, COM interfaces can get much larger. And in case there are more than one interface of a target object, most probably all of them must be implemented.

If you choose to modify vtable, it requires you, as a developer, to be a much more attentive while keeping in mind a number of details. The number of lines of code you will have to write to be able to manage multiple source object instances of the same interface or to call target's methods increases correspondingly. On the other hand, this approach doesn't have the restrictions that the proxy object approach does, i.e. you do not need to perform hooking right after target's creation (this can be done at any time), and if there are methods you don't need to intercept, there is no need to implement hooks for them.

Get the described sample source code: [C++ sample source](#)