

# Red Teaming's Dojo

## Playing around COM objects - PART 1

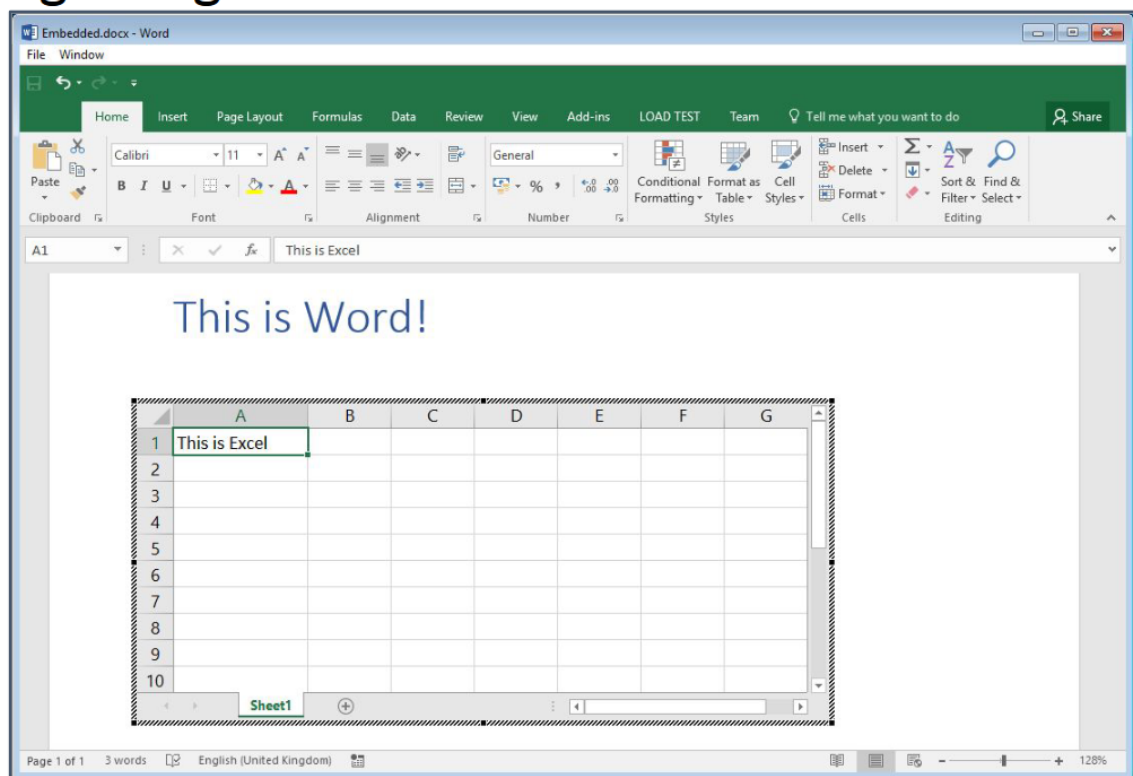
Component Object Model

### Introduction

COM is a platform-independent, distributed, object-oriented system **for creating binary software components that can interact**. COM is the foundation technology for Microsoft's OLE (Object Linking and Embedding)(compound documents) and ActiveX (Internet-enabled components) technologies.

In other word, COM was originally created to enable Microsoft Office applications to communicate and exchange data between documents such embedding an Excel chart inside a Word document or PowerPoint presentation <---This ability called OLE.

### In the Beginning was OLE



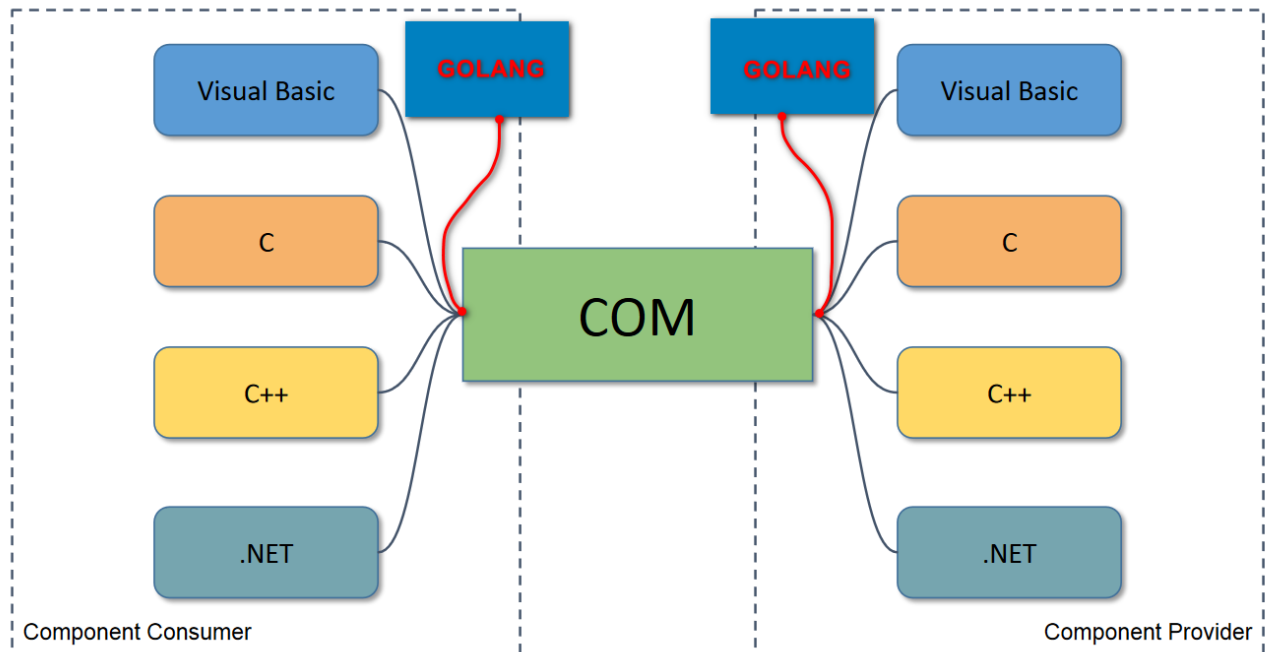
OLE ability this figure was taken from the talk of James Forshaw "COM in 60 seconds"

COM objects can be created with a variety of programming languages. Object-oriented languages, such as C++ *and the challenge for me was how I could interact with COM objects using GOLANG via Win32 API functions devoted for this.* 😊

As well as COM designed **to promote software interoperability**; that is, to allow two or more applications or “components” to easily interact with one another, even if they were written by different vendors at different

times, in different programming languages, or if they are running on different machines and different operating systems.

## Interoperability Heaven



### COM objects and Interfaces

First of all, what is an **object**? An object is an **instantiation** of some **class**. At a generic level, a “class” is the definition of a set of related attributes and methods grouped together for a specified purpose. The purpose is generally to provide some service to “things” outside the object, namely clients that want to make use of those services.

**A client never has direct access to the COM object in its entirety.** Instead, clients always access the object through clearly **defined contracts**: the interfaces that the object supports.

A COM object exposes its features through an **interface**, which is a collection of member functions.

An **interface** is basically an **abstract class**, containing only **pure virtual functions** and has no data members.

Therefore, **an abstract class is a class that contain only pure virtual declaration of functions** (the “ = 0 ” indicates the purity)) and has no data members:

```
1 class IClassA
2 {
3 public:
4     virtual void func1() = 0;
5 };
```

An abstract class is a class that **cannot be used to create objects**; however, it can be subclassed.

So, the following line of code is not valid and it will throw an error as an object cannot be created for the

abstract class:

```
1 IClassA* pNewInstance = new IClassA;
```

**The interface class can only be approached by using a pointer to the virtual table that exposes the methods in the interface.** Interface does not come by itself, it usually comes with an inherited class that implements the exposed method in the interface. Such a class that implements the interface exposed methods is often called a co-class(derived class). Here is an example of a co-class:

```
1 class ClassB: public IClassA
2 {
3     public:
4         virtual void func1() { // implementation here }
5 };
```

**Code Explanation:** Here, **IClassA** is the base class, and as it has a pure virtual function (**func1**), it has become an abstract class. **ClassB** is derived from the parent class **IClassA**. The **func1** is defined in the derived class.

We can use a global method to create an instance of the co-class or we can use a static method as well.

**The technique of using a method that creates an instance of a co-class and returning a pointer to its interface is often called Class Factoring.** Here is the global create instance method:

```
1 IClassA* CreateInstance()
2 {
3     return new ClassB;
4 }
```

In the main function, we try to create a pointer of base class type, it will be created successfully, and we can point it to the derived class. This pointer can be used to call the derived class function.

```
1 int main() {
2     //IClassA b; ----- > this line will throw an error
3     IClassA* b = CreateInstance(); //----- > pointer can be created, so this line is correct
4     b -> func1();
5 }
```

**A COM class** is identified by using a unique **128-bit** Class ID (**CLSID**) that associates a class with a particular deployment in the file system, which for Windows is a DLL or EXE. A CLSID is a **GUID**, which means that no other class has the same CLSID.(*Strongly-typed*)

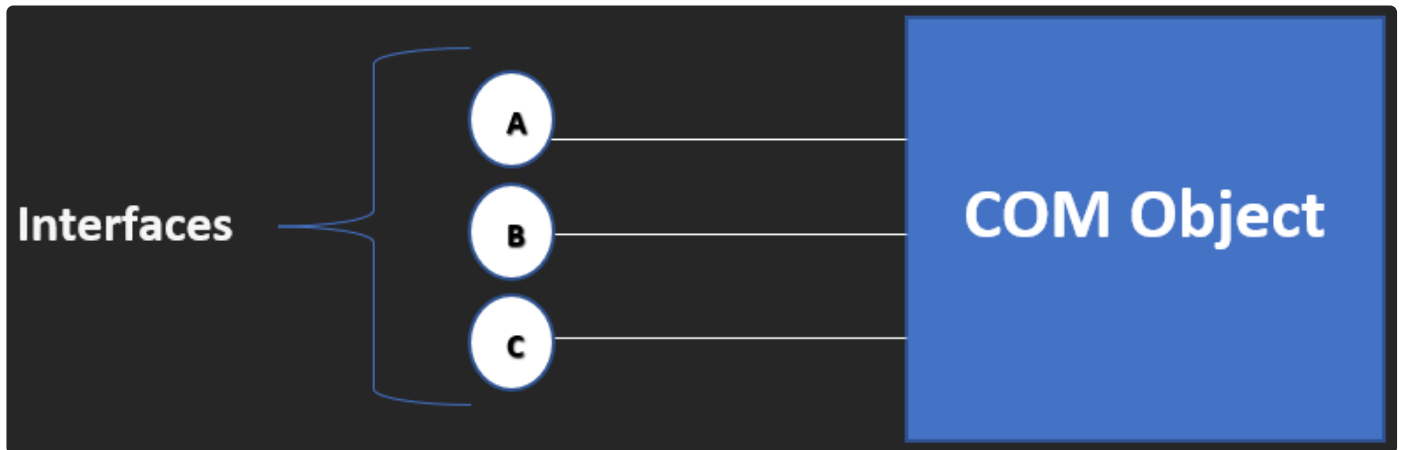
Also we can use a programmatic identifier (**ProgID**) **which is a registry entry that can be associated with a CLSID**. Like the CLSID, the ProgID identifies a class but with less precision because it is not guaranteed to be globally unique.(*it's human readable and are locally scoped*)

**Interfaces** are strongly typed. Every interface has its own unique interface identifier, named an **IID**, which eliminates collisions that could occur with human-readable names. The IID is a globally unique identifier (**GUID**). *The name of an interface is always prefixed with an "I" by convention.*

**i** **COM Clients only interact with pointers to interfaces as we described before:** When a client has access to an object, it has nothing more than a pointer through which it can access the functions in the interface. The pointer is opaque, meaning that it hides all aspects of internal implementation. In COM, the client can only call functions of the interface to which it has a **pointer**.

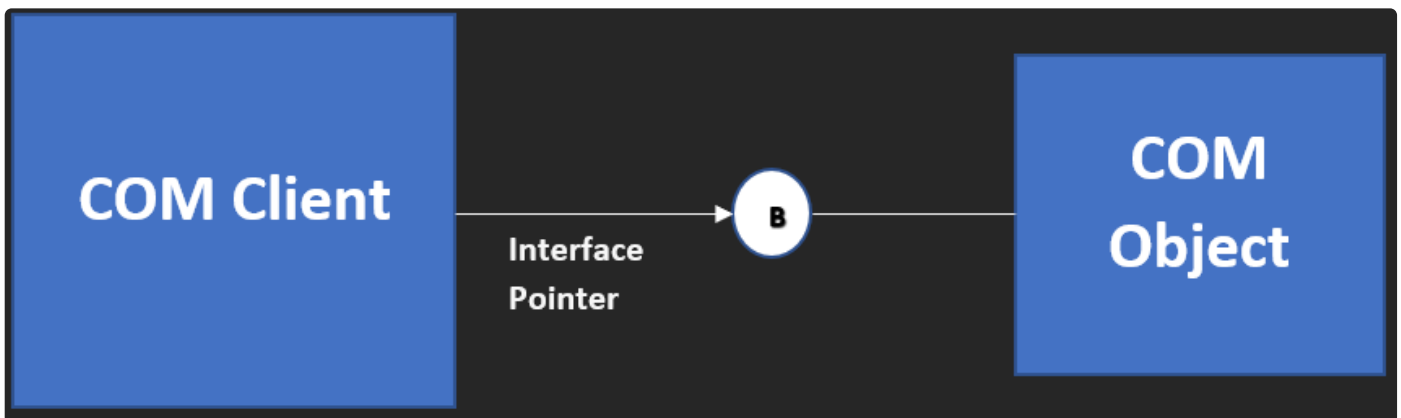
## Object Presentation

It is convenient to use a simple presentation to show the relation between a COM object and its interfaces :



Typical picture of a COM object that supports 3 interfaces A,B, and C

Now, when a client want to interact with this COM object he must use a pointer to the targeted interface:

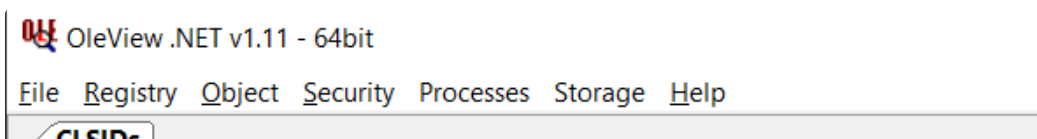


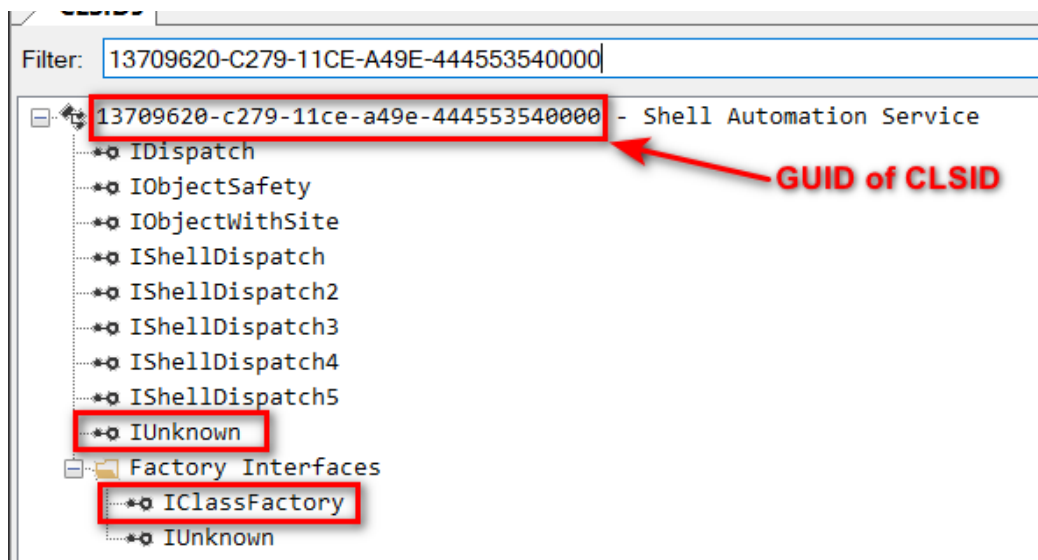
Interface B extend towards the client connected to it

In order to enumerate COM objects through a number of different views (e.g. by CLSID, by ProgID, by server executable), enumerate interfaces on the object and then create an instance and invoke methods we will use the famous tool [OleViewDotNet](#) made by [James Forshaw](#) and the tool [COMView](#).

Let's try to analyze this CLSID that have the following GUID :

**13709620-C279-11CE-A49E-444553540000,**





13709620-C279-11CE-A49E-444553540000

In COM, an object can support multiple interfaces as depicted on the above picture: IDispatch, IObjectSafety, IObjectWithSite, IShellDispatch{1,2,3,4,5} and there are 2 interfaces that should demands a little special attention : **IUnknown & IClassFactory**.

As we said previously an interface is strongly-typed which mean it has an only unique identifier called **IID**:

OleView .NET v1.11 - 64bit

File Registry Object Security Processes Storage Help

CLSIDs 13709620-c279-11ce-a...			
CLSID	Supported Interfaces	Type	Library
Interfaces: Refresh			
Name	IID	Methods	VTable Offset
IObjectSafety	CB5BDC81-93C1-11CF-8F20-00805F2CD064	3	shell32.dll+0x599760
IObjectWithSite	FC4801A3-2BA9-11CF-A229-00AA003D7352	5	shell32.dll+0x599720
IShellDispatch	D8F015C0-C278-11CE-A49E-444553540000	3	shell32.dll+0x599788
IShellDispatch2	A4C6892C-3BA9-11D2-9DEA-00C04FB16162	3	shell32.dll+0x599788
IShellDispatch3	177160CA-BB5A-411C-841D-BD38FACDEAA0	3	shell32.dll+0x599788
IShellDispatch4	EFD84B2D-4BCF-4298-BE25-EB542A59FBDA	3	shell32.dll+0x599788
IShellDispatch5	866738B9-6CF2-4DE8-8767-F794EBE74F4E	3	shell32.dll+0x599788
IUnknown	00000000-0000-0000-C000-000000000046	3	shell32.dll+0x599788
Factory Interfaces:			
Name	IID	Methods	VTable Offset
IClassFactory	00000001-0000-0000-C000-000000000046	3	shell32.dll+0x5AD2C8
IUnknown	00000000-0000-0000-C000-000000000046	3	shell32.dll+0x5AD2C8

Supported interfaces on the CoClass CLSID 13709620-C279-11CE-A49E-444553540000

## 1) IUnknown interface

IUnknown is a fundamental interface in COM that contains basic operations of not only all objects, but all interfaces as well. An object is not considered a COM object unless it implements this interface.

All interfaces in COM are **polymorphic** with IUnknown:

```

C Unknwn.h x
C: > Program Files (x86) > Windows Kits > 10 > Include > 10.0.22000.0 > um > C Unknwn.h > ...

107 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////
108 // IID_IUnknown and all other system IIDs are provided in UUID.LIB
109 // Link that library in with your proxies, clients and servers
110 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////
111
112 #if (_MSC_VER >= 1100) && defined(__cplusplus) && !defined(CINTERFACE)
113     EXTERN_C const IID IID_IUnknown;
114     extern "C++"
115     {
116         MIDL_INTERFACE("00000000-0000-0000-C000-000000000046")
117         IUnknown
118         {
119         public:
120             BEGIN_INTERFACE
121             virtual HRESULT STDMETHODCALLTYPE QueryInterface(
122                 /* [in] */ REFIID riid,
123                 /* [iid_is][out] */ _COM_Outptr_ void __RPC_FAR * __RPC_FAR *ppvObject) = 0;
124
125             virtual ULONG STDMETHODCALLTYPE AddRef( void) = 0;
126
127             virtual ULONG STDMETHODCALLTYPE Release( void) = 0;
128
129             template<class Q>
130             HRESULT
131             #ifdef _M_CEE_PURE
132                 _clrcall
133             #else
134                 STDMETHODCALLTYPE
135             #endif
136             QueryInterface(_COM_Outptr_ Q** pp)
137             {
138                 return QueryInterface(__uuidof(Q), (void **)pp);
139             }
140
141             END_INTERFACE
142         };

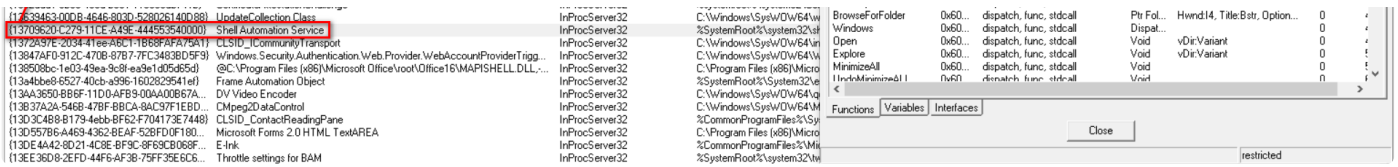
```

The definition of IUnknown using the IDL notation.

So if you look at the first 3 functions in any interfaces you will see **QueryInterface**, **AddRef**, and **Release**:

The screenshot shows the COMView application. The main window displays a list of TypeLibs with columns for CLSID, Name, GUID, TypeKind, and Flags. A red box highlights the entry for IShellDispatch6 (GUID: 66958641-3801-4621-B393-444553540000). A red arrow points from this entry to a secondary window titled 'TypeInfo IShellDispatch6 [Updated IShellDispatch]'. This window shows a table of interface methods:

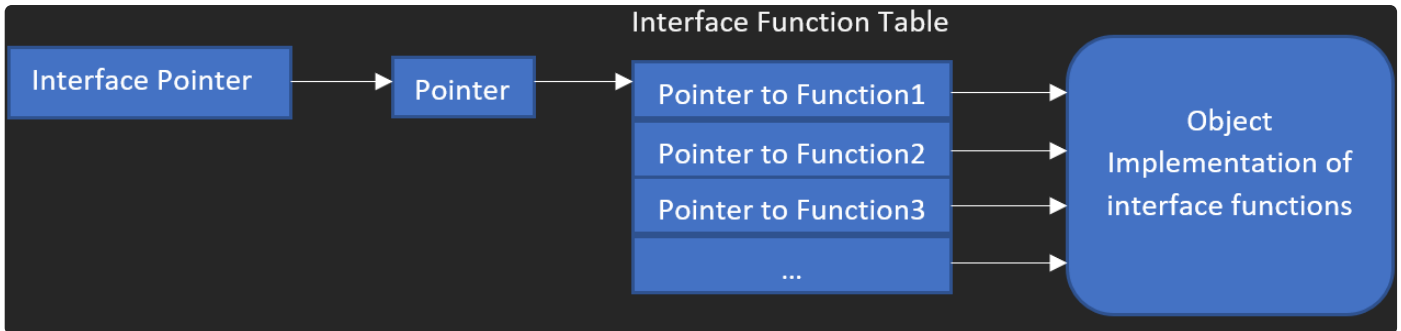
Name	menid	FuncKind	InvKind	CallConv	rcType	Params	FL	ots/vtz
QueryInterface	Dx60...	dispatch	func	stdcall	Void	riid Ptr GUID, ppvObj Ptr...	1	
AddRef	Dx60...	dispatch	func	stdcall	UI4		1	
Release	Dx60...	dispatch	func	stdcall	UI4		1	
GetTypeInfoCount	Dx60...	dispatch	func	stdcall	Void	pTypeInfo Ptr UInt	1	
GetTypeInfo	Dx60...	dispatch	func	stdcall	Void	riInfo UInt, lcid UI4, ppInfo...	1	
GetIDsOfNames	Dx60...	dispatch	func	stdcall	Void	riid Ptr GUID, rgids Names...	1	
Invoke	Dx60...	dispatch	func	stdcall	Void	dispId Member I4, riid Ptr G...	1	
Application	Dx60...	dispatch	propertyget	stdcall	Disp...		0	
Parent	Dx60...	dispatch	propertyget	stdcall	Disp...		0	
NameSpace	Dx60...	dispatch	func	stdcall	Ptr Fol...	vDir Variant	0	



The re-usage of the 3 functions of IUnknown interface in IShellDispatch5 interface due to polymorphic concept.

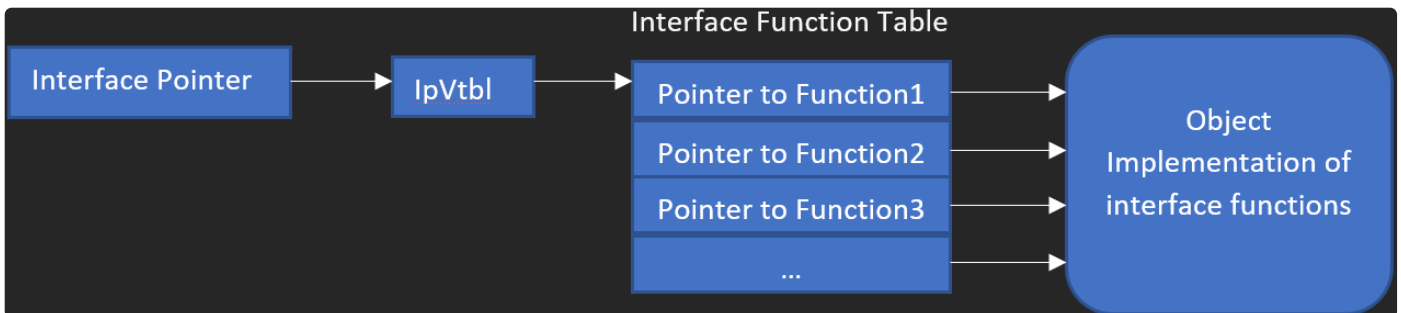
**In other words, IUnknown is the base interface from which all other interfaces inherit.**

Before continuing, I would like to highlight something we've discussed it before which is this expression "COM Client only interact with a **pointer to interfaces**", this pointer is a pointer to an array of pointers to the objects's implementations of the interface member functions:



A client has a pointer to an interface which is a pointer to a pointer to an array (table) of pointers to the object's implementation.

By convention the pointer to the interface function table is called the **pVtbl** pointer. The table itself is generally referred to with the name **Vtbl** for "virtual function table.":



Convention places object data following the pointer to the interface function table.

## 2) IClassFactory interface

As you know, we create instances of a COM object using:

- The interface specifications.
- The CLSID declaration in the COM class.

**We do not directly use the COM class to create an instance of a type of COM object.**

Instead, **an intermediate object called a class object is used to create instances of a COM object.**

### Class Object

A class object is a specialized type of COM object that knows how to create a specific type of COM object.



There is a one-to-one relationship between a **class object** and a **COM object**.

Every type of class object knows how to create only one type of COM object. For example, if two COM objects, called **01** and **02**, are implemented within a COM server, two class objects must also be implemented, one that knows how to create **01** and one that knows how to create **02**.

A class object can be considered of as a "**creator**" object. Its only goal is to **create instances of a COM object**.

Clients obtain a pointer to a class object's interface by asking the COM subsystem for a specific class object that can create the COM object they want. Using this interface pointer, clients have the class object create one or more instances of their associated COM object.

The COM specification defines a COM object creation interface called **IClassFactory**:

```
IClassFactory : public IUnknown
{
public:
    virtual /* [local] */ HRESULT STDMETHODCALLTYPE CreateInstance(
        /* [annotation][unique][in] */
        _In_opt_ IUnknown *pUnkOuter,
        /* [annotation][in] */
        _In_ REFIID riid,
        /* [annotation][iid_is][out] */
        _COM_Outptr_ void **ppvObject) = 0;

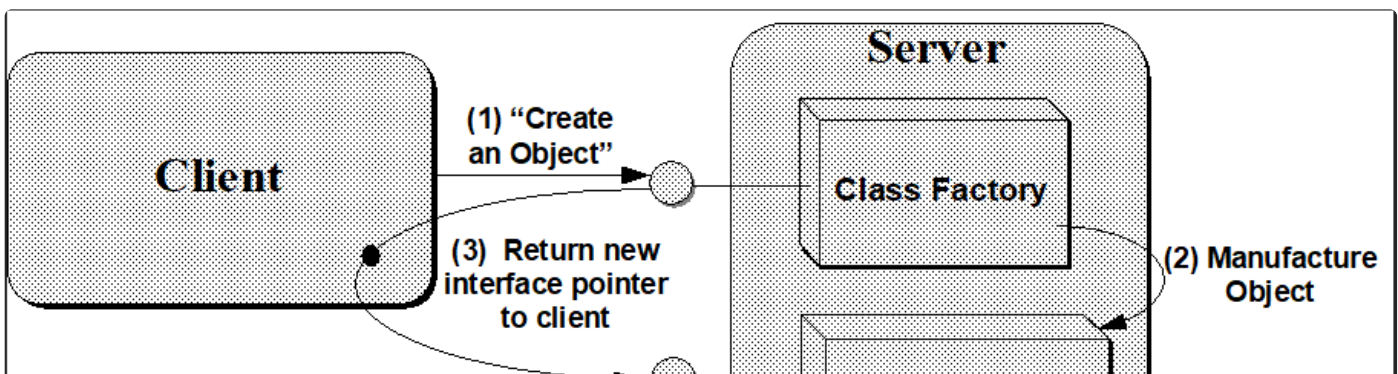
    virtual /* [local] */ HRESULT STDMETHODCALLTYPE LockServer(
        /* [in] */ BOOL fLock) = 0;
};
```

IClassfactory implementation

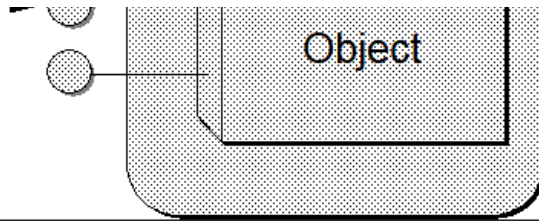
Class objects that implement **IClassFactory** as their object creation interface are called class factories.

Given a CLSID the client must now create an object of that class in order to make use of its services. It does so using two steps:

1. Obtain the "class factory" for the CLSID.
2. Ask the class factory to instantiate an object of the class
3. Returning an interface pointer to the client.







A client asks a class factory in the server to create an object.

- i The IClassFactory interface is implemented by COM servers on a “class factory” object for the purpose of creating new objects of a particular class.

## OI32.dll!CoGetClassObject

Now that we grasp what a class factory we can examine how a client obtains the class factory.

This COM library function do whatever is necessary to obtain a class factory object for the given CLSID and return one of that class factory's interface pointers to the client. After that the client may calls IClassFactory::CreateInstance to instantiate objects of the class.

```
1 HRESULT CoGetClassObject(  
2     [in]          REFCLSID rclsid,  
3     [in]          DWORD    dwClsContext,  
4     [in, optional] LPVOID   pvReserved,  
5     [in]          REFIID   riid,  
6     [out]         LPVOID   *ppv  
7 );
```

Let's spot the most important parameters :

[in] rclsid

The **CLSID** associated with the data and code that you will use to create the objects.

[in] dwClsContext

The **context in which the executable code is to be run**. To enable a remote activation, include CLSCTX\_REMOTE\_SERVER. For more information on the context values and their use, see the [CLSCTX](#) enumeration.

[in] riid

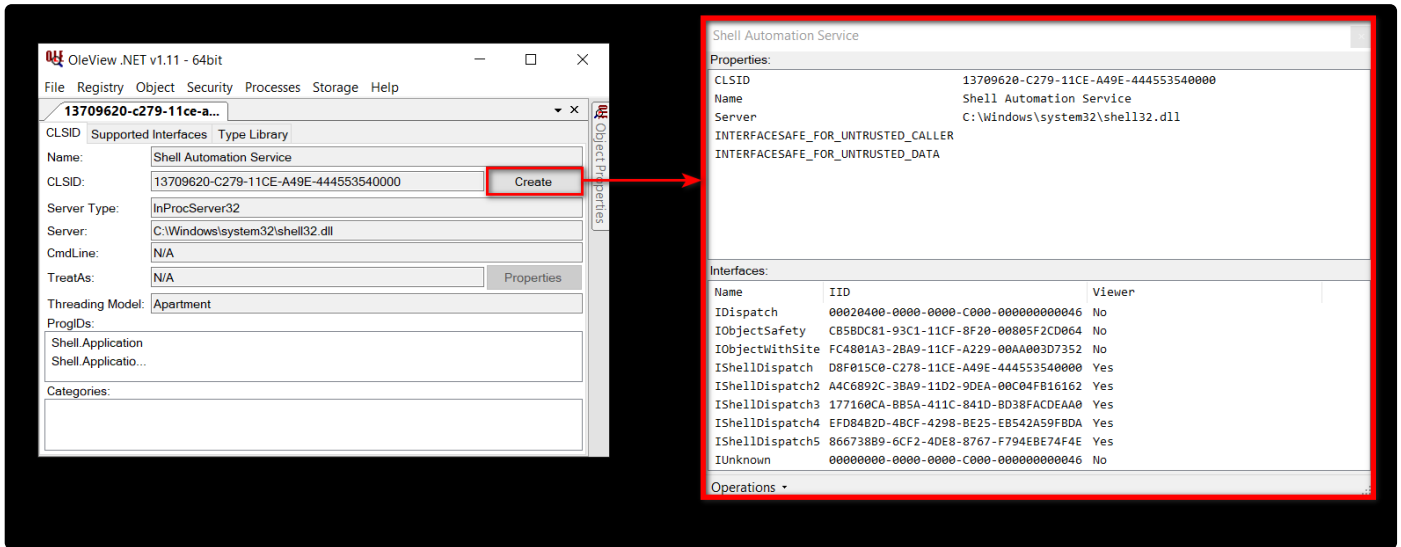
**Reference to the identifier of the interface**, which will be supplied in *ppv* on successful return. **This interface will be used to communicate with the class object**. Typically this value is **IID\_IClassFactory**, although other values such as IID\_IClassFactory2 which supports a form of licensing are allowed.

[out] ppv

The **address of pointer variable that receives the interface pointer requested in riid**. Upon successful return, **\*ppv contains the requested interface pointer**.

This function return **S\_OK** if location and connection to the specified class object was successful.

Now, we will try to instantiate an object of the class {13709620-C279-11CE-A49E-444553540000} using **OleViewDotNet** which seems for me very abstract :



instantiate an object of the class {13709620-C279-11CE-A49E-444553540000}

## Hooking class factoring

In order to understand what's under the hood I decided to hook class factoring routine and practice what [Mr.Un1k0d3r](#) teach us, so let's take a more insightful look through WinDBG at how the calls are chained together. If we ask OLE32 about all the **CoGet\*** and **CreateI\*** used by **OleViewDotNet** :

```
1 0:008> x OLE32!CoGet*
2 00007ffb`2ffd4790 ole32!CoGetInterceptor (struct _GUID *, struct IUnknown *, struct _GUID *
3 00007ffb`2ff8e180 ole32!CoGetObject (wchar_t *, struct tagBIND_OPTS *, struct _GUID *, voi
4 00007ffb`2ffc3830 ole32!CoGetSystemWow64DirectoryW (wchar_t *, unsigned int)
5 00007ffb`2ffa3350 ole32!CoGetInterceptorFromTypeInfo (struct _GUID *, struct IUnknown *, s
6 00007ffb`2ff91070 ole32!CoGetInterceptorForOle32 (struct _GUID *, struct IUnknown *, struc
7 0:008> x OLE32!CreateI*
8 00007ffb`2ffc38e0 ole32!CreateILockBytesOnHGGlobalStub (void *, int, struct ILockBytes **)
9 00007ffb`2ff84c30 ole32!CreateItemMoniker (wchar_t *, wchar_t *, struct IMoniker **)
10
```

As you can notice above **OleViewDotNet** don't use `Ole32.dll!CoGetClassObject`, and you may encounter **CoCreateInstance** which is simply a wrapper function for `CoGetClassObject` and `IClassFactory`.

By googling little a bit I found this precious information mentioned on MSDN documentation that said:

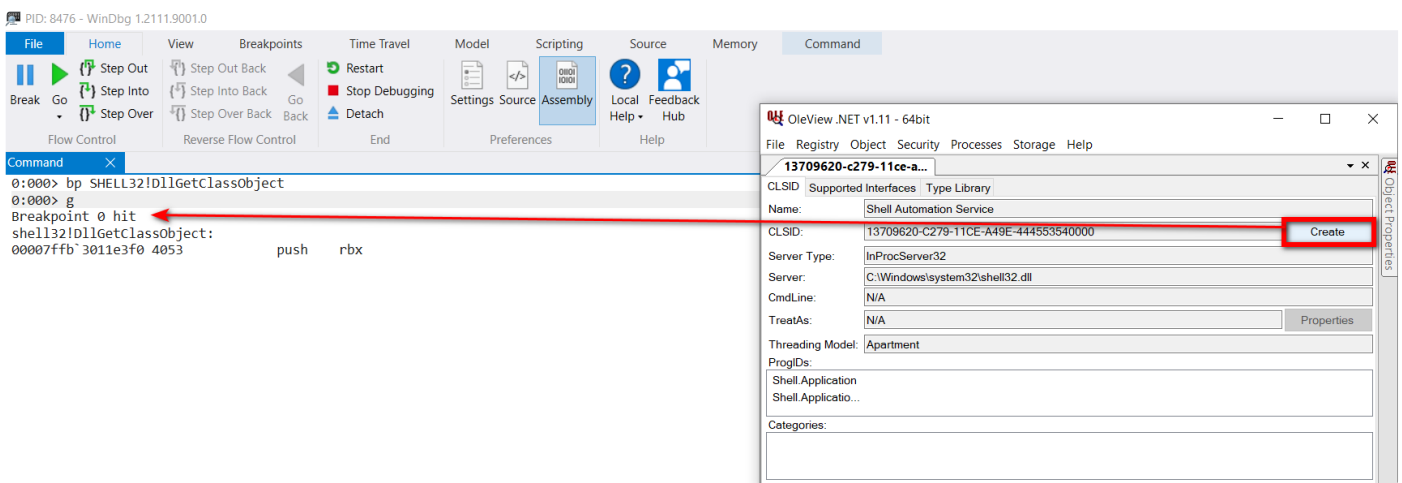
**i** You should not call **DllGetClassObject** directly. When an object is defined in a DLL, **CoGetClassObject** calls the **CoLoadLibrary** function to load the DLL (in our case this dll is `shell32.dll`), which, in turn, calls **DllGetClassObject**.

We can also verify the existence of this function through WinDBG :

```
1 0:000> x
2 00007ffb`3011e3f0 shell32!DllGetClassObject (void)
```

So we place a breakpoint at **SHELL32!DllGetClassObject** and let it run until we hit it:

```
1 0:000> bp SHELL32!DllGetClassObject
2 0:000> g
3 Breakpoint 0 hit
4 shell32!DllGetClassObject:
5 00007ffb`3011e3f0 4053          push    rbx
```



We can so confirm that **OleViewDotNet** is using **SHELL32!DllGetClassObject** for instantiating an object of the class with CLSID : {13709620-C279-11CE-A49E-444553540000}

To gain a full picture of what's under the hood, we can use the nice (Trace and watch utility)'**wt -l 2**' WinDBG command to gain a two-level-depth hierarchical function call :

```
1 0:007> bp SHELL32!DllGetClassObject
2 0:007> g
3 Breakpoint 0 hit
4 shell32!DllGetClassObject:
5 00007ffb`3011e3f0 4053          push    rbx
6 0:000> wt -l 2
7 Tracing shell32!DllGetClassObject to return address 00007ffb`30df50cb
8 491    0 [ 0] shell32!DllGetClassObject
9 6     0 [ 1] shell32!_security_check_cookie
10 498   6 [ 0] shell32!DllGetClassObject
11
12 504 instructions were executed in 503 events (0 from other threads)
13
14 Function Name                               Invocations  MinInst  MaxInst  AvgInst
15 shell32!DllGetClassObject                    1           498     498     498
16 shell32!_security_check_cookie               1           6       6       6
17
```

```

18 0 system calls were executed
19
20 combase!CClassCache::CDllPathEntry::GetClassObject+0x29 [inlined in combase!CClassCache::C
21 00007ffb`30df50cb f605ae182a0002 test byte ptr [combase!Microsoft_Windows_COM_PerfEnab
22

```

From the above result I deduced that **SHELL32!DllGetClassObject** is calling **combase!DllGetClassObject** thus I added an other breakpoint to confirm that result :

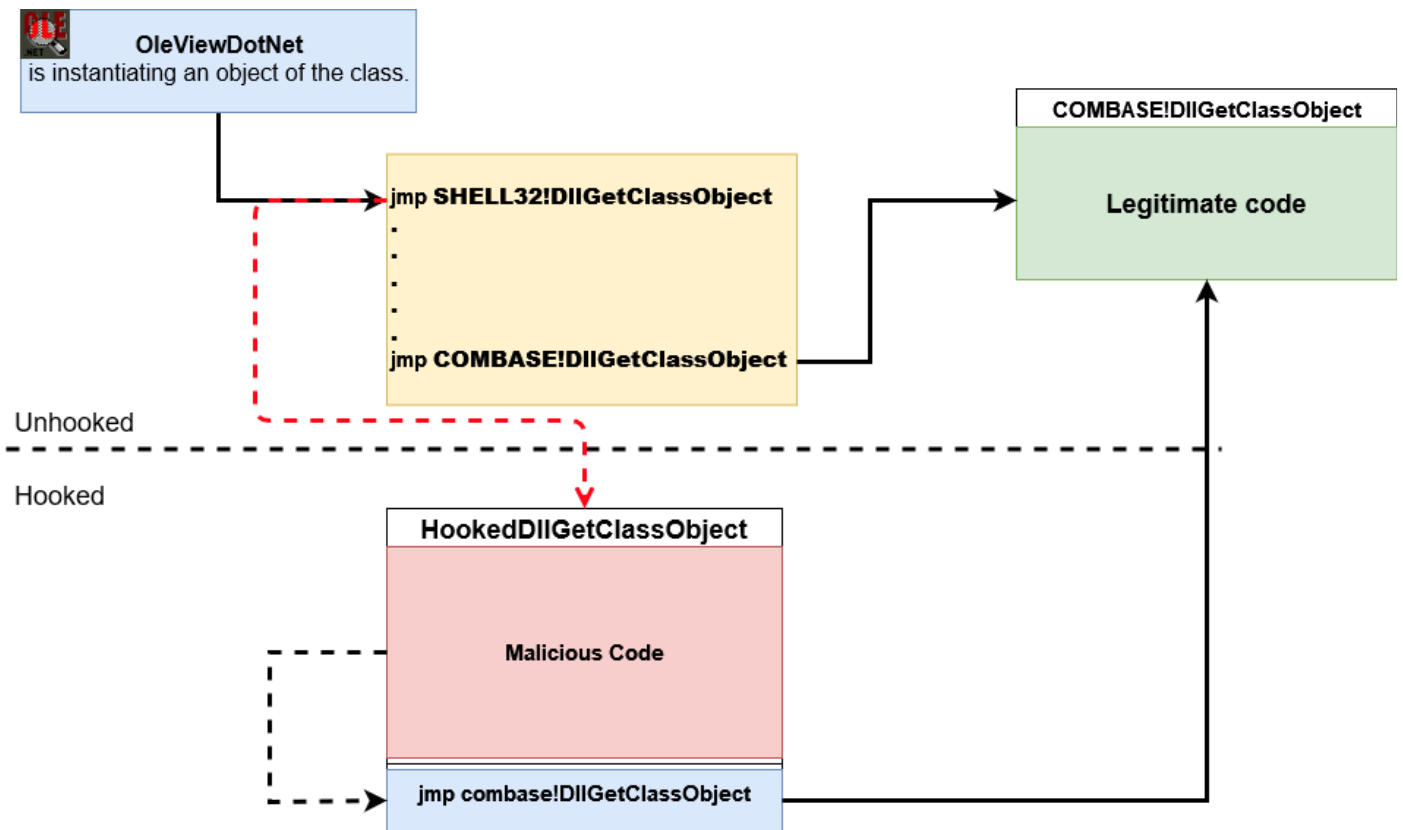
```

1 0:009> bl
2      0 e Disable Clear 00007ffb`3011e3f0 0001 (0001) 0:**** shell32!DllGetClassObject
3      1 e Disable Clear 00007ffb`30e1ff90 0001 (0001) 0:**** combase!DllGetClassObject
4 0:009> g
5 Breakpoint 0 hit
6 shell32!DllGetClassObject:
7 00007ffb`3011e3f0 4053          push    rbx
8 0:000> g
9 Breakpoint 1 hit
10 combase!DllGetClassObject:
11 00007ffb`30e1ff90 48895c2408  mov    qword ptr [rsp+8],rbx ss:000000cf`9a6fc730=ffffff

```

Now I have a clear idea how I will craft my hooker 😊

Below is a simplified diagram that aims to visualize the flow of events before and after **SHELL32!DllGetClassObject** is hooked :



Before hooking

Based on my debugging analysis when OleViewDotNet want to instantiate an object of the class mainly it jumps to **SHELL32!DllGetClassObject** then executing an other jump to **combase!DllGetClassObject** :

The screenshot shows WinDbg at PID 15688. The command window shows the following sequence of events:

```

0:000> g
Breakpoint 0 hit
shell32!DllGetClassObject: 1 call
00007fff`1897e490 4053          push    rbx
0:000> g
Breakpoint 1 hit
combase!DllGetClassObject: 2 call
00007fff`1a4cff90 48895c2408      mov     qword ptr [rsp+8],rbx ss:000000f9`ed30cab0=0401cd0100003
0:000> g

```

The Shell Automation Service Properties window shows the following details:

- CLSID: 13709620-C279-11CE-A49E-444553540000
- Name: Shell Automation Service
- Server: C:\Windows\system32\shell32.dll
- INTERFACESAFE\_FOR\_UNTRUSTED\_CALLER: Yes
- INTERFACESAFE\_FOR\_UNTRUSTED\_DATA: Yes

A red text overlay in the center of the Shell Automation Service window reads: "Objet of the class successfully is instantiate".

Before moving to after hooking section let me show you the **DllGetClassObject** definition based on MSDN documentation :

```

1 HRESULT DllGetClassObject(
2     [in] REFCLSID rclsid, //The CLSID that will associate the correct data and code.
3     [in] REFIID riid, //Usually, this is IID_IClassFactory, a reference to the identifier
4     [out] LPVOID *ppv //The address of a pointer variable that receives the interface pointer
5 );

```

If you have already noticed that **CoGetClassObject** parameters look like **DllGetClassObject**.

### After hooking

- OleViewDotNet calls **shell32!DllGetClassObject** like before hooking.
- OleViewDotNet looks up **shell32!DllGetClassObject** address but here is the magic, we patched dynamically this address to a malicious address that point on a rogue **comhook!HookedDllGetClassObject** function:

```

1 0:004> g
2 Breakpoint 0 hit
3 shell32!DllGetClassObject:
4 00007fff`1897e490 b8fa138469      mov     eax,offset comhook!HookedDllGetClassObject (00007fff`1a4c
5 0:000> u 00007fff`1897e490
6 shell32!DllGetClassObject:
7 00007fff`1897e490 b8fa138469      mov     eax,offset comhook!HookedDllGetClassObject (00007fff`1a4c
8 00007fff`1897e495 ffe0           jmp     rax
9 00007fff`1897e497 56           push   rsi
10 00007fff`1897e498 4157         push   r15
11 00007fff`1897e49a 4881ecb0020000 sub    rsp,2B0h
12 00007fff`1897e4a1 488b05d06f6600 mov    rax,qword ptr [shell32!_security_cookie (00007fff`1a4c
13 00007fff`1897e4a8 4833c4       xor    rax,rsp
14 00007fff`1897e4ab 4889842490020000 mov    qword ptr [rsp+290h],rax

```

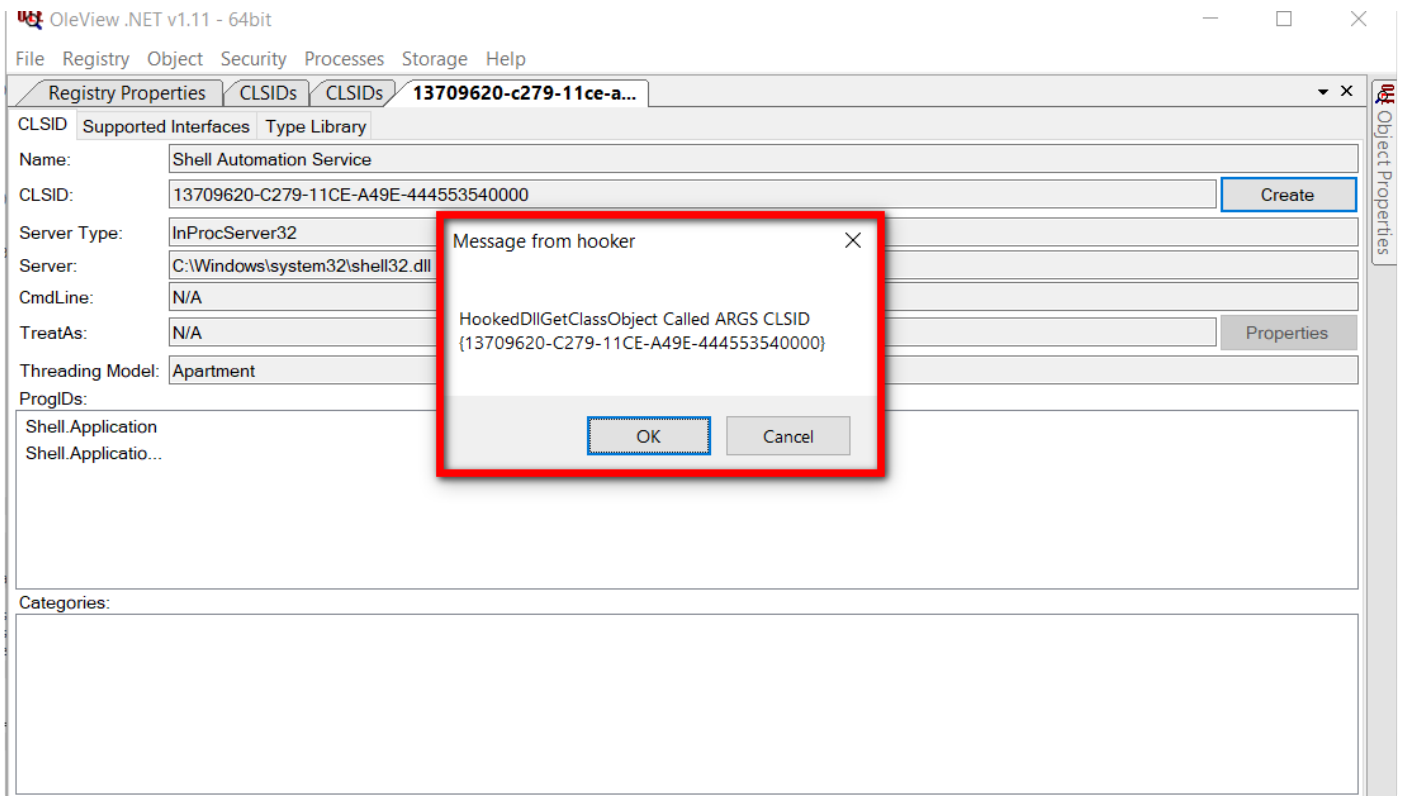
Take a look on line **7** and **8**, the address of **HookedDllGetClassObject** is **0x698413fa**, and if we disassemble it we will land into our malicious code and for poc purpose, **comhook!HookedDllGetClassObject** intercepts the **rclsid** parameter and execute pop up message holding the CLSID of the COM class :

```

1 0:000> uf 00000000`698413fa
2 comhook!HookedDllGetClassObject:
3 00000000`698413fa 55          push    rbp
4 00000000`698413fb 4889e5     mov     rbp,rsb
5 00000000`698413fe 4883ec40   sub     rsp,40h
6 00000000`69841402 48894d10   mov     qword ptr [rbp+10h],rcx
7 00000000`69841406 48895518   mov     qword ptr [rbp+18h],rdx
8 00000000`6984140a 4c894520   mov     qword ptr [rbp+20h],r8
9 00000000`6984140e ba00010000 mov     edx,100h
10 00000000`69841413 b940000000 mov     ecx,40h
11 00000000`69841418 488b05057e0000 mov     rax,qword ptr [comhook!rdgco+0x190c (00000000`69841418)]
12 00000000`6984141f ffd0      call   rax
13 00000000`69841421 488945f8   mov     qword ptr [rbp-8],rax
14 00000000`69841425 488d45e8   lea    rax,[rbp-18h]
15 00000000`69841429 4889c2     mov     rdx,rax
16 00000000`6984142c 488b4d10   mov     rcx,qword ptr [rbp+10h]
17 00000000`69841430 488b05fd7e0000 mov     rax,qword ptr [comhook!rdgco+0x1a1c (00000000`69841430)]
18 00000000`69841437 ffd0      call   rax
19 00000000`69841439 488b55e8   mov     rdx,qword ptr [rbp-18h]
20 00000000`6984143d 488b45f8   mov     rax,qword ptr [rbp-8]
21 00000000`69841441 4989d1     mov     r9,rdx
22 00000000`69841444 4c8d05b52b0000 lea    r8,[comhook!Hook+0x2b5f (00000000`69844000)]
23 00000000`6984144b baff000000 mov     edx,0FFh
24 00000000`69841450 4889c1     mov     rcx,rax
25 00000000`69841453 e858ffff   call   comhook+0x13b0 (00000000`698413b0)
26 00000000`69841458 488b45f8   mov     rax,qword ptr [rbp-8]
27 00000000`6984145c 41b901000000 mov     r9d,1
28 00000000`69841462 4c8d05c62b0000 lea    r8,[comhook!Hook+0x2b8e (00000000`6984402f)]
29 00000000`69841469 4889c2     mov     rdx,rax
30 00000000`6984146c b900000000 mov     ecx,0
31 00000000`69841471 488b05cc7e0000 mov     rax,qword ptr [comhook!rdgco+0x1a2c (00000000`69841471)]
32 00000000`69841478 ffd0      call   rax
33 00000000`6984147a 488d0597640000 lea    rax,[comhook!rdgco (00000000`69847918)]
34 00000000`69841481 488b00     mov     rax,qword ptr [rax]
35 00000000`69841484 488b4d20   mov     rcx,qword ptr [rbp+20h]
36 00000000`69841488 488b5518   mov     rdx,qword ptr [rbp+18h]
37 00000000`6984148c 4989c8     mov     r8,rcx
38 00000000`6984148f 488b4d10   mov     rcx,qword ptr [rbp+10h]
39 00000000`69841493 ffd0      call   rax
40 00000000`69841495 8945f4 uf     mov     dword ptr [rbp-0Ch],eax
41 00000000`69841498 8b45f4     mov     eax,dword ptr [rbp-0Ch]
42 00000000`6984149b 4883c440   add     rsp,40h
43 00000000`6984149f 5d        pop     rbp
44 00000000`698414a0 c3        ret

```





3. **comhook!HookedDllGetClassObject** at the end call **combase!DllGetClassObject** routine at line **39** and if we can confirm that by adding a breakpoint at the address **0x69841478**:

```

1 0:000> g
2 Breakpoint 2 hit
3 comhook!HookedDllGetClassObject+0x99:
4 00000000`69841493 ffd0          call     rax {combase!DllGetClassObject (00007fff`1a4cff90)

```

### comhook.dll-x64

Frankly, I included this exercise in my research first to get familiar with known win32 API functions used to create COM objects and also simulate what Anti-malware and EDR software often utilize to intercept suspicious calls by injecting their **dlls** into processes and make some checks.

In our context, I inject comhook.dll into OleViewDotNet that allows me to understand what's under the hood in the class factoring phase of any COM object creation.

Let me share the snippet code of my comhook.dll and explain the most relevant part of it :

Global variables :

```

1 BOOL hooked = FALSE;
2 typedef BOOL (WINAPI * DLLMAIN)( HINSTANCE, DWORD, LPVOID );
3 typedef HRESULT (*RealDllGetClassObject)(REFCLSID, REFIID, LPVOID*);
4 RealDllGetClassObject rdgco;
5 FARPROC shell32DllGetClassObject;

```

- (i) The C programming language provides a keyword called **typedef**, which you can use to give a type a new name.

## HookedDllGetClassObject()

```
1 HRESULT HookedDllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv){
2
3     CHAR *log = (CHAR*)GlobalAlloc(GPTR, 256);
4     HRESULT hr;
5     LPOLESTR os;
6     StringFromCLSID(rclsid, &os);
7     snprintf(log, 255, "HookedDllGetClassObject Called ARGS CLSID %ws\n", os);
8     MessageBox(0,log,"Message from hooker",1);
9     hr = rdgco(rclsid, riid, ppv);
10    return hr;
11 }
```

## Hook()

```
1 VOID Hook()
2 {
3     if(!hooked) {
4         //Get the address of shell32!DllGetClassObject
5         shell32DllGetClassObject = GetProcAddress(LoadLibrary("shell32.dll"), "DllGetClassObject");
6         //Get the address of combase!DllGetClassObject
7         rdgco = (RealDllGetClassObject)GetProcAddress(LoadLibrary("combase.dll"), "DllGetClassObject");
8
9         DWORD dwSize = 7;
10        DWORD dwOld = 0;
11        //Allocates the specified number of bytes from the heap.
12        CHAR *patch = (CHAR*)GlobalAlloc(GPTR, dwSize);
13        //Doing casting stuff
14        CHAR *addr = (CHAR*)shell32DllGetClassObject;
15        long long longHookedDllGetClassObject = (long long)HookedDllGetClassObject;
16        DWORD dwHooked = (DWORD)longHookedDllGetClassObject;
17        //Changes the protection of shell32DllGetClassObject addresss to be able to patch
18        VirtualProtect((VOID*)shell32DllGetClassObject, dwSize, PAGE_EXECUTE_READWRITE, &dwOld);
19
20        //The tricky part
21        DWORD i = 0;
22        DWORD position = 1;
23        patch[0] = 0xb8;
24        for(i; i < 4; i++) {
25            CHAR current = dwHooked;
26            patch[position++] = current;
27            dwHooked >>= 8;
28        }
29        patch[5] = 0xff;
30        patch[6] = 0xe0;
```

```

31
32     //patch the address of shell32DllGetClassObject to point into HookedDllGetClassObj
33     memcpy(addr, patch, dwSize);
34     VirtualProtect((VOID*)shell32DllGetClassObject, dwSize, PAGE_EXECUTE_READ, &dwOld)
35 }
36
37 hooked = TRUE;
38 }

```

I will explain the tricky part, roughly the idea is to patch the address of **shell32DllGetClassObject** to an address that will allow us to jump into **HookedDllGetClassObject**.

+ The local variable **addr** hold the address of **shell32DllGetClassObject**.

+ The local variable **dwHooked** hold the address of **HookedDllGetClassObject** which is **0x698413fa** which is represented **in 4 bytes**.

Now, we want that the patch point us in an address that perform the following instruction:

```

root@osboxes:~/tmp# python OpAsm.1.3.py 64 asm "mov eax, 0x698413fa; jmp rax"
OpAsm Tools v1.3 / Mr.Un1k0d3r RingZer0 Team

ASSEMBLY OUTPUT
0:  b8 fa 13 84 69      mov    eax,0x698413fa
5:  ff e0              jmp    rax

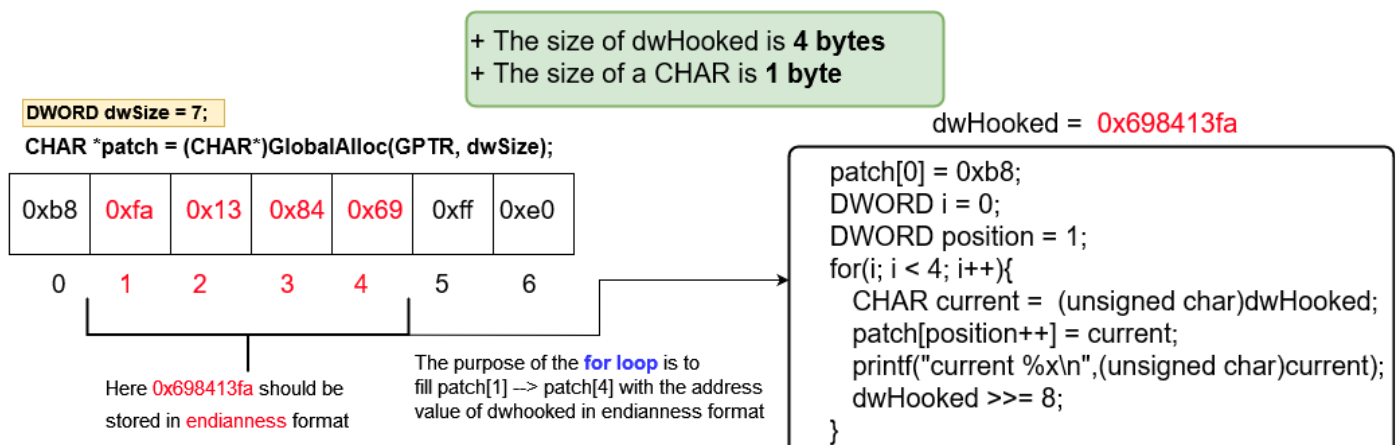
```

Based on the above result our asm code is represented in **7 opcodes**, thus we fix **dwSize** to **7** and we allocate memory for the CHAR\* variable **patch** by this instruction :

```
1 CHAR *patch = (CHAR*)GlobalAlloc(GPTR, dwSize);
```

**i** Keep in mind that when pushing stuff on the stack there is endianness format that should be take on consideration while storing **0x698413fa** on that stack.

The following figure explain what's happened in the tricky part



```
patch[5] = 0xff;
patch[6] = 0xe0;
```

First iteration : i = 0

CHAR current = 0x698413fa;

Since the size of a char is 1 byte so automatically **current** will store last value : 0xfa

patch[position++] = current;

Here **position ++** pass the value then increment --> patch[1] = 0xfa

dwHooked >>= 8;

Here we **right shift** 8 bits(1 byte) the value 0x698413fa

At the end of the first iteration dwhooked = 0x00698413 and after we proceed the same logic for i=1 ..... i = 3

Finally, we will patch the address of **shell32DllGetClassObject** by the address of **patch** variable through this instruction : memcpy(addr, patch, dwSize);

## COM via C(not C++)

The purpose of this section will be the implementation of those tasks in C language:

1. Obtain the "class factory" for our targeted CLSID. --> through **DllGetClassObject**.
2. Ask the class factory to instantiate an object of the class --> through **CreateInstance** method.
3. Get ShellExecute ID.
4. Initialize parameters to be used on Invoke method.
5. Pop the calc.

**1) COM object creation**

**2) get a pointer into IShellDispatch2**

**3) Invoke ShellExecute method by setting file parameter to calc**

Invoke ShellExecute

Name	Type	Value	Dir	Optional
File	System.String	calc	in	No
vArgs	System.Object	<null>	in	Yes
vDir	System.Object	<null>	in	Yes
vOperation	System.Object	<null>	in	Yes
vShow	System.Object	<null>	in	Yes

OleViewDotNet pop calc through ShellExecute method


**Goal : Reach what OleViewDotNet did through C language.**

## C vs C++

COM is based on a *binary interoperability* standard, rather than a *language interoperability* standard. Any language supporting “**structure**” or “**record**” types containing **double-indirected access to a table of function pointers** is suitable.

That being said, COM can declare interface declarations for both C++ and C . The C++ definition of an interface, which in general is of the form:

```
1 interface ISomeInterface{
2     virtual RET_T MemberFunction(ARG1_T arg1, ARG2_T arg2 /*, etc */);
3     [Other member functions]
4     ...
5     };
```

 Did you know that a struct can store a pointer to some function?

then the corresponding C declaration of that interface looks like :

```
1 typedef struct ISomeInterface
2     {
3     ISomeInterfaceVtbl * pVtbl;
4     } ISomeInterface;
5
6 typedef struct ISomeInterfaceVtbl ISomeInterfaceVtbl;
7 struct ISomeInterfaceVtbl
8     {
9     RET_T (*MemberFunction)(ISomeInterface * this, ARG1_T arg1,
10         ARG2_T arg2 /*, etc */);
11     [Other member functions]
12     } ;
```

What we've done above is to recreate a C++ class, using plain C. The **ISomeInterface** struct is really a C++ class. A C++ class is really nothing more than a struct whose **first member is always a pointer to its VTable** (an array of function pointers) -- an array that contains pointers to all the functions inside of that class. **The first argument passed to an object's function is a pointer to the object (struct) itself. (This is referred to as the hidden " this " pointer.)**

### 1. Obtain the “class factory” for our targeted CLSID. --> through DllGetClassObject

Before a program can use any COM object, it must initialize COM, which is done by calling the function **Colnitalize**. This need be done only once, so a good place to do it is at the very start of the program.

```
1 DEFINE_GUID(clsid, 0x13709620, 0xc279, 0x11ce, 0xa4, 0x9e, 0x44, 0x45, 0x53, 0x54, 0x00, 0:
```

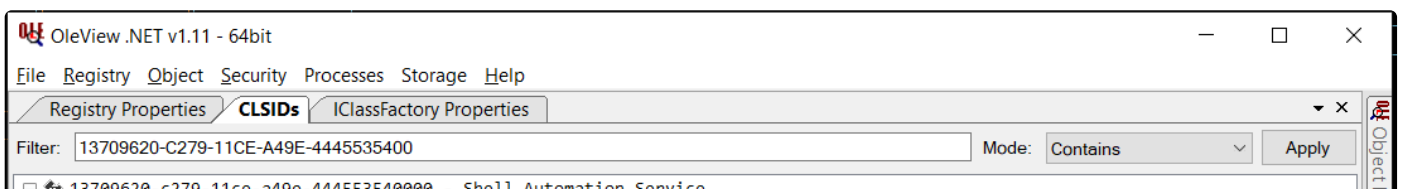
The above is a macro. A `#define` in one of the Microsoft include files allows your compiler to compile the

above into a 16 byte array.

Next, the program calls `DllGetClassObject` to get a pointer to `shell32.dll`'s `IClassFactory` object. Note that we pass the `CLSID` object's GUID as the first argument. We also pass a pointer to our variable `icf` which is where a pointer to the `IClassFactory` will be returned to us, if all goes well:

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <initguid.h>
4 #include <stdint.h>
5 //gcc .\comfunc.c -o com.exe -lole32 -luuid -loleaut32
6
7 //The CLSID {13709620-C279-11CE-A49E-4445535400} associated with the data and code that we
8 DEFINE_GUID(clsid, 0x13709620, 0xc279, 0x11ce, 0xa4, 0x9e, 0x44, 0x45, 0x53, 0x54, 0x00, 0x00, 0x00, 0x00)
9
10 int main(char argc, char **argv){
11
12     LPOLESTR clsidstr = NULL;
13     StringFromCLSID(&clsid, &clsidstr);
14     printf("Our targeted CLSID is %ls\n", clsidstr);
15     HRESULT hr;
16     hr = CoInitialize(NULL);
17
18     FARPROC DllGetClassObject = GetProcAddress(LoadLibrary("shell32.dll"), "DllGetClassObject");
19     printf("DllGetClassObject is at 0x%p\n\n", DllGetClassObject);
20
21     IClassFactory *icf = NULL;
22     // Get shell32.DLL's IClassFactory
23     hr = DllGetClassObject(&clsid, &IID_IClassFactory, (void **)&icf);
24
25     if(hr != S_OK) {
26         printf("DllGetClassObject failed to do something. Error %d HRESULT 0x%08x\n", hr, hr);
27
28         CoUninitialize();
29         ExitProcess(0);
30     }
31     //For debugging purposes
32     HMODULE shell32address = GetModuleHandle("shell32.dll");
33     printf("shell32.dll address is :%p \tIClassFactory's Vtable address is:%p \n", shell32address, DllGetClassObject);
34     uint64_t val = (uint64_t)icf->lpVtbl - (uint64_t)shell32address;
35     printf("The offset is 0x%p - 0x%p = 0x%llx", icf->lpVtbl, shell32address, val);
36     return 0;
37 }
```

Now, if we compile the above code and run it, we will get a match between the resulted offset and the one shown in **OleViewDotNet** which prove that we have now a pointer to the VTable of `IClassFactory` :





The screenshot shows the OleView.NET interface with the 'Factory Interfaces' folder expanded to show 'IUnknown'. A tooltip for 'IUnknown' displays the following information:

- Name: IUnknown
- IID: 00000000-0000-0000-C000-000000000046
- VTable Address: shell32.dll+0x5B02D8

Below the interface view, a terminal window shows the following commands and output:

```
PS C:\Users\t3nb3w\Desktop\COMFUN> gcc .\comfunc.c -o com.exe -lole32 -luuid
PS C:\Users\t3nb3w\Desktop\COMFUN> .\com.exe
Our targeted CLSID is {13709620-C279-11CE-A49E-444553540000}
DllGetClassObject is at 0x00007fff1897e490

shell32.dll address is :00007fff18910000      IClassFactory's Vtable address is:00007fff18ec02d8
The offset is 0x00007fff18910000 - 0x00007fff18ec02d8 = 0x5b02d8
PS C:\Users\t3nb3w\Desktop\COMFUN> █
```

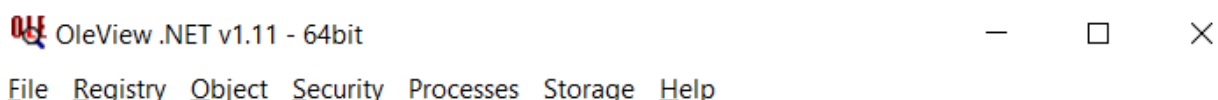
## 2. Ask the class factory to instantiate an object of the class through CreateInstance method

Once we have the `IClassFactory` object, we can call its `CreateInstance` function to get a `IDispatch` object. Note how we use the `IClassFactory` to call its `CreateInstance` function. We get the function via `IClassFactory`'s VTable. Also note that we pass the `IClassFactory` pointer as **the first argument**.

Note that we pass `IDispatch`'s VTable GUID as **the third argument**. And for **the fourth argument**, we pass a pointer to our variable `id` which is where a pointer to an `IDispatch`'s object will be returned to us, if all goes well:

```
1 // Create an IDispatch object
2 IDispatch *id = NULL;
3 hr = icf->lpVtbl->CreateInstance(icf, NULL, &IID_IDispatch, (void **)&id);
4 if(hr != S_OK) {
5     printf("CreateInstance failed to do something. Error %d HRESULT 0x%08x\n", GetE
6
7     CoUninitialize();
8     ExitProcess(0);
9 }
10 printf("[+]IDispatch's Vtable address is:%p \n",id->lpVtbl);
11 uint64_t val1 = (uint64_t)id->lpVtbl - (uint64_t)shell32address;
12 printf("The offset IDispatch is 0x%p - 0x%p = 0x%llx\n", id->lpVtbl , shell32address ,
```

Again compile and run :



Registry Properties | CLSIDs | 13709620-c279-11ce-a49... | 13709620-c279-11ce-a... | X

CLSID | Supported Interfaces | Type Library

Interfaces: Refresh

Name	IID	Methods	VTable Offset
IDispatch	00020400-0000-0000-C000-000000000046	7	shell32.dll+0x59C788
IObjectSafety	CB5BDC81-93C1-11CF-8F20-00805F2CD064	3	shell32.dll+0x59C760
IObjectWithSite	FC4801A3-2BA9-11CF-A229-00AA003D7352	5	shell32.dll+0x59C720
IShellDispatch	D8F015C0-C278-11CE-A49E-444553540000	3	shell32.dll+0x59C788
IShellDispatch2	A4C6892C-3BA9-11D2-9DEA-00C04FB16162	3	shell32.dll+0x59C788
IShellDispatch3	177160CA-BB5A-411C-841D-BD38FACDEAA0	3	shell32.dll+0x59C788
IShellDispatch4	EFD84B2D-4BCF-4298-BE25-EB542A59FBDA	3	shell32.dll+0x59C788

Factory Interfaces:

Name	IID	Methods	VTable Offset
IClassFactory	00000001-0000-0000-C000-000000000046	3	shell32.dll+0x5B02D8
IUnknown	00000000-0000-0000-C000-000000000046	3	shell32.dll+0x5B02D8

```
PS C:\Users\t3nb3w\Desktop\COMFUN> .\com.exe
Our targeted CLSID is {13709620-C279-11CE-A49E-444553540000}
DllGetObject is at 0x00007fff1897e490

shell32.dll address is :00007fff18910000
[+]IClassFactory's Vtable address is:00007fff18ec02d8
The offset IClassFactory is 0x00007fff18ec02d8 - 0x00007fff18910000 = 0x5b02d8
[+]IDispatch's Vtable address is:00007fff18eac788
The offset IDispatch is 0x00007fff18eac788 - 0x00007fff18910000 = 0x59c788
```

Object Properties

**i** IDispatch interface usage : Exposes objects, methods and properties to programming tools and other applications that support Automation.

### 3. Get ShellExecute ID

Once we have the IDispatch object, we can call its GetIDsOfNames function to get the COM dispatch identifier (DISPID) of ShellExecute .

Note that we pass the IDispatch pointer as the first argument. We pass a reference of IID\_NULL as the second argument(Reserved for future use. Must be IID\_NULL.), for the third argument, we pass ShellExecute as WCHAR to be mapped, for the fourth argument the count of the names to be mapped(1 in our case), for the fifth argument the locale context in which to interpret the names. And last argument we pass the address of dispid which is the id value of ShellExecute that will be returned to us, if all goes well:

```
1 // get function ID
```

```

2  WCHAR *member = L"ShellExecute";
3  DISPID dispid = 0;
4
5  hr = id->lpVtbl->GetIDsOfNames(id, &IID_NULL, &member, 1, LOCALE_USER_DEFAULT, &dispid);
6  if(hr != S_OK) {
7      printf("GetIDsOfNames failed to do something. Error %d HRESULT 0x%08x\n", GetL
8
9      CoUninitialize();
10     ExitProcess(0);
11 }
12
13 printf("DISPID 0x%08x\n", dispid);

```

Again compile and run :

The screenshot shows two windows from the COMView tool. The top window, 'TypeLib Shell32 [Microsoft Shell Controls And Automation]', lists several interfaces. The bottom window, 'Typelib IShellDispatch2 [Updated IShellDispatch]', shows a list of methods. The 'ShellExecute' method is highlighted in blue, and its memory address '0x60030001' is circled in red. A red arrow points from this address to the 'DISPID 0x60030001' line in the terminal output below.

Name	GUID	TypeKind	Flags	Functions	Variables	Interfa...	Idx
ShellFolderView	{62112AA1-E...	COCLASS (5)	CanCreate	0	0	2	22
ShellFolderViewOptions	{742A99A0-C...	ENUM (0)		0	7	0	23
IShellDispatch	{D8F015C0-C...	DISPATCH (4)	Hidden, D...	30	0	1	24
IShellDispatch2	{A4C6892C-3...	DISPATCH (4)	Hidden, D...	39	0	1	25

Name	memid	FuncKind,InvKind,CallCo...	rcType	Params
FindFiles	0x60020013	dispatch, func, stdcall	Void	
FindComputer	0x60020014	dispatch, func, stdcall	Void	
RefreshMenu	0x60020015	dispatch, func, stdcall	Void	
ControlPanelItem	0x60020016	dispatch, func, stdcall	Void	bstrDir:Bstr
IsRestricted	0x60030000	dispatch, func, stdcall	I4	Group:Bstr, Restriction:Bstr
ShellExecute	0x60030001	dispatch, func, stdcall	Void	File:Bstr, vArgs:Variant, v...
FindPrinter	0x60030002	dispatch, func, stdcall	Void	Name:Bstr, location:Bstr, ...

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
PS C:\Users\t3nb3w\Desktop\COMFUN> .\com.exe
Our targeted CLSID is {13709620-C279-11CE-A49E-444553540000}
DllGetClassObject is at 0x00007fff1897e490

shell32.dll address is :00007fff18910000
[+]IClassFactory's Vtable address is:00007fff18ec02d8
The offset IClassFactory is 0x00007fff18ec02d8 - 0x00007fff18910000 = 0x5b02d8
[+]IDispatch's Vtable address is:00007fff18eac788
The offset IDispatch is 0x00007fff18eac788 - 0x00007fff18910000 = 0x59c788
DISPID 0x60030001

```

4.Initialize parameters to be used on Invoke method

According to MSDN, **Invoke** goal is to provide access to properties and methods exposed by an object and this is what we need to pop up the calc :

```

1 HRESULT Invoke(
2     [in]     IDispatch *this
3     [in]     DISPID   dispIdMember,
4     [in]     REFIID   riid,
5     [in]     LCID     lcid,
6     [in]     WORD     wFlags,
7     [in, out] DISPPARAMS *pDispParams,
8     [out]    VARIANT   *pVarResult,
9     [out]    EXCEPINFO *pExcepInfo,
10    [out]    UINT      *puArgErr
11 );

```

In our situation, note that we pass the `IDispatch` pointer as **the 1 arg**. Then, we pass the **dispid** id of **ShellExecute** as **the 2 arg**. We pass a reference of `IID_NULL` as **the 3 arg** (Reserved for future use. Must be `IID_NULL`.), for **the 4 arg** the locale context in which to interpret the names, for **the 5 arg** flags describing the context of the **Invoke** call (in our case we pass `DISPATCH_METHOD`).

**The 6 arg** pointer to a `DISPPARAMS` structure containing **an array of arguments** passed **ShellExecute**, an array of argument DISPIDs for named arguments, and **counts** for the number of elements in the arrays.

```

1 typedef struct tagDISPPARAMS {
2     VARIANTARG *rgvarg; //An array of arguments.
3     DISPID     *rgdispidNamedArgs;
4     UINT       cArgs; //The number of arguments.
5     UINT       cNamedArgs;
6 } DISPPARAMS;

```

**i** **VARIANTARG** describes arguments passed within `DISPPARAMS`, and **VARIANT** to specify variant data that cannot be passed by reference.

In our case, we need only to set up 2 elements to reach our goal :

```

C++

typedef struct tagVARIANT {
    union {
        struct {
            VARTYPE vt;
            WORD    wReserved1;
            WORD    wReserved2;
            WORD    wReserved3;


```

```

WORD        wReserved3;
union {
    LONGLONG    l1Val;
    LONG        lVal;
    BYTE        bVal;
    SHORT       iVal;
    FLOAT      fltVal;
    DOUBLE      dblVal;
    VARIANT_BOOL boolVal;
    VARIANT_BOOL __OBSOLETE__VARIANT_BOOL;
    SCODE       scode;
    CY         cyVal;
    DATE       date;
    BSTR       bstrVal;
    IUnknown   *punkVal;
}

```

Before continuing let me highlight some stuff regarding BSTR(Basic string or binary string):

 A BSTR (Basic string or binary string) is a string data type that is used by COM, Automation, and Interop functions. Use the BSTR data type in all interfaces that will be accessed from script.

A BSTR is a composite data type that consists of a **length prefix**, a **data string**, and a **terminator**. For more details check : <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/automat/bstr>

```

1 // initialize parameters
2
3 //VARIANT describes arguments passed within DISPPARAMS.
4 VARIANT args = { VT_EMPTY };
5 args.vt = VT_BSTR;
6 args.bstrVal = SysAllocString(L"calc");
7
8 // Contains the arguments passed to ShellExecute method.
9 DISPPARAMS dp = {&args, NULL, 1, 0};

```

At this stage, we prepared all the necessary ingredients to pop the calc using the invoke method; for the rest of the arguments that I did not mention just take a look on: <https://docs.microsoft.com/en-us/windows/win32/api/oidl/nf-oidl-idispatch-invoke>

## 5. Pop the calc

```

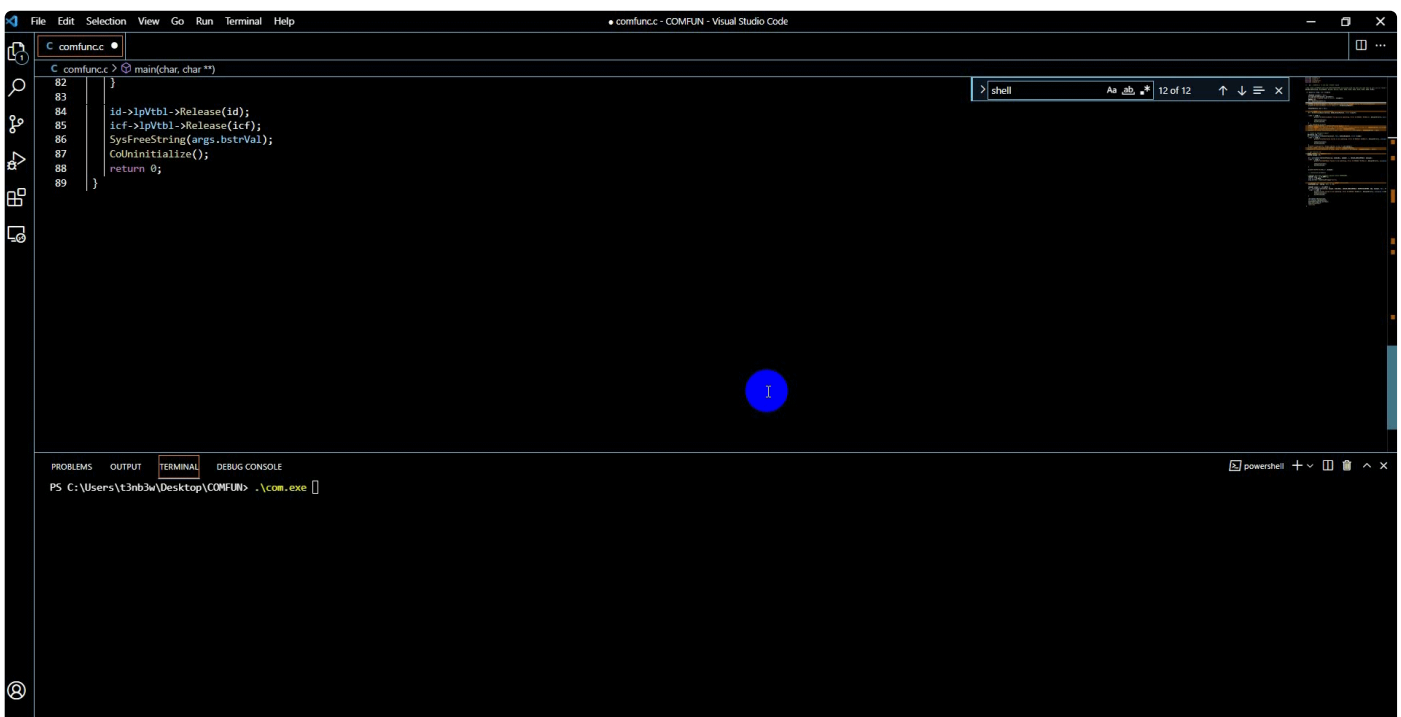
1  VARIANT output = { VT_EMPTY };
2  hr = id->lpVtbl->Invoke(id, dispid, &IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dp,
3      if(hr != S_OK) {
4          printf("Invoke failed to do something. Error %d HRESULT 0x%08x\n", GetLastError
5              CoUninitialize();
6              ExitProcess(0);
7          }
8
9      id->lpVtbl->Release(id);
10     icf->lpVtbl->Release(icf);
11     SysFreeString(args.bstrVal);
12     CoUninitialize();

```

So next, we call the `IDispatch`'s `Release` and `IClassFactory`'s `Release` functions. Once we do this, our `id` and `icf` variables no longer contains a valid pointer to anything. It's garbage now. We call also `SysFreeString` to deallocate a string allocated previously.

Finally, we must call `CoUninitialize` to allow COM to clean up some internal stuff. This needs to be done once only, so it's best to do it at the end of our program (but only if `CoInitialize` succeeded).

Demo:



## Conclusion

As you notice the blog was long, but I truly share with you all the things I learned regarding the internal of COM which represent a fundamental pillar for Windows as OS. (COM is a big ocean of course I did not cover all things like single thread apartment, proxy/stub, security aspects, I promise my next parts will be about that).



Since I'm very obsessed with Windows especially when I'm facing in security community like [James FORSHAW](#) talks regarding COM, also [Matt Nelson's](#) blog that explain lateral movements through DCOM... as well as some AVs that expose misconfigure COM object that led to LPE and self-bypass a poc was done by [Denis Skvortcov](#).

What I mentioned in the previous paragraph, combined with the fact that one day I tried to implement WMI using golang, but I failed that represents the reasons why I wrote this blog since I knew nothing about COM, this is how it works, and now we need to learn theory and basic stuff, also slow down and seek to deeply understand the topic. This patience will be fruitful since you can now explore and develop stuff with creativity.

**Final Note:** I am not a Windows Internal expert I'm just a learner, If you think I said anything incorrect anywhere, feel free to reach out to me and correct me, I would highly appreciate that. And finally, thank you very much for taking your time to read this post.

---

## References

GitHub - Seggaeman/DeveloperWorkshopCOMATL3: Companion CD content for the book "D...  
GitHub

COM in plain C  
CodeProject

The Component Object Model - Win32 apps  
docsmsft

Abstract Class in C++ | Implementation of Constructor & Destructor  
EDUCBA

COM Interface Basics  
CodeProject

COM classes, Objects, Factories

CoGetClassObject function (combaseapi.h) - Win32 apps  
docsmsft

DllGetClassObject function (combaseapi.h) - Win32 apps  
docsmsft

Home - RingZero CTF