Finding Interactive User COM Objects using PowerShell

tiraniddo.dev/2018/09/finding-interactive-user-com-objects_9.html

Easily one of the most interesting blogs on Windows behaviour is Raymond Chen's The Old New Thing. I noticed he'd recently posted about using "Interactive User" (IU) COM objects to go from an elevated application (in the UAC sense) to the current user for the desktop. What interested me is that registering arbitrary COM objects as IU can have security consequences, and of course this blog entry didn't mention anything about that.

The two potential security issues can be summarised as:

1. An IU COM object can be a sandbox escape if it has non-default security (for example Project Zero Issue 1079) as you can start a COM server outside the sandbox and call methods on the object.
2. An IU COM object can be a cross-session elevation of privilege if it has non-default security (for example Project Zero Issue 1021) as you can start a COM server in a different console session and call methods on the object.

I've blogged about this before when I discuss how I exploited a reference cycle bug in *NtCreateLowBoxToken* (see Project Zero Issue 483) and discussed how to use my OleView.NET tool find classes to check. Why do I need another blog post about it? I recently uploaded version 1.5 of my OleView.NET tool which comes with a fairly comprehensive PowerShell module and this seemed like a good opportunity on doing a quick tutorial on using the module to find targets for analysis to see if you can find a new sandbox escape or cross session exploit.

Note I'm not discussing how you go about reverse engineering the COM implementation for anything we find. I also won't be dropping any unknown bugs, but just giving you the information needed to find interesting COM servers.

Getting Started with PowerShell Module

First things first, you'll need to grab the release of v1.5 from the THIS LINK (edit: you can now also get the module from the PowerShell Gallery). Unpack it to a directory on your system then open PowerShell and navigate to the unpacked directory. Make sure you've allowed arbitrary scripts to run in PS, then run the following command to load the module.

```
PS C:\> Import-Module .\OleViewDotNet.psd1
```

As long as you see no errors the PS module will now be loaded. Next we need to capture a database of all COM registration information on the current machine. Normally when you open the GUI of OleView.NET the database will be loaded automatically, but not in the module. Instead you'll need load it manually using the following command:

```
PS C:\> Get-ComDatabase -SetCurrent
```

The *Get-ComDatabase* cmdlet parses the system configuration for all COM information my tool knowns about. This can take some time (maybe up to a minute, more if you have Process Monitor running), so it'll show a progress dialog. By specifying the *-SetCurrent* parameter we will store the database as the current global database, for the current session. Many of the commands in the module take a *-Database* parameter where you can specify the database you want to extract information from. Ensuring you pass the correct value gets tedious after a while so by setting the current database you never need to specify the database explicitly (unless you want to use a different one).

Now it's going to suck if every time you want to look at some COM information you need to run the lengthy *Get-ComDatabase* command. Trust me, I've stared at the progress bar too long. That's why I implemented a simple save and reload feature. Running the following command will write the current database out to the file *com.db:*

```
PS C:\> Set-ComDatabase .\com.db
```

You can then reload using the following command:

```
PS C:\> Get-ComDatabase .\com.db -SetCurrent
```

You'll find this is significantly faster. Worth noting, if you open a 64 bit PS command line you'll capture a database of the 64 bit view of COM, where as in 32 bit PS you'll get a 32 bit view.

Finding Interactive User COM Servers

With the database loaded we can now query the database for COM registration information. You can get a handle to the underlying database object as the variable *$comdb* using the following command:

```
PS C:\> $comdb = Get-CurrentComDatabase
```

However, I wouldn't recommend using the COM database directly as it's not really designed for ease of use. Instead I provide various cmdlets to extract information from the database which I've summarised in the following table:

Command	Description
Get-ComClass	Get list of registered COM classes
Get-ComInterface	Get list of registered COM interfaces
Get-ComAppId	Get list of registered COM AppIDs
Get-ComCategory	Get list of registered COM categories
Get-ComRuntimeClass	Get list of Windows Runtime classes
Get-ComRuntimeServer	Get list of Windows Runtime servers

Each command defaults to returning all registered objects from the database. They also take a range of parameters to filter the output to a collection or a single entry. I'd recommend passing the name of the command to *Get-Help* to see descriptions of the parameters and examples of use.

Why didn't I expose it as a relational database, say using SQL? The database is really an object collection and one thing PS is good at is interacting with objects. You can use the *Where-Object* command to filter objects, or *Select-Object* to extract certain properties and so on. Therefore it's probably a lot more work to build on a native query syntax that just let you write PS scripts to filter, sort and group. To make life easier I have spent some time trying to link objects together, so for example each COM class object has an *AppIdEntry* property which links to the object for the AppID (if registered). In turn the AppID entry has a *ClassEntries* property which will then tell you all classes registered with that AppID.

Okay, let's get a list of classes that are registered with *RunAs* set to *"Interactive User"*. The class object returned from *Get-ComClass* has a *RunAs* property which is set to the name of the user account that the COM server runs as. You also need to only look for COM servers which run out of process, we can do this by filtering for only *LocalServer32* classes.

Run the following command to do the filtering:

```
PS C:\> $runas = Get-GetComClass -ServerType LocalServer32 | ? RunAs -eq "Interactive
User"
```

You should now find the *$runas* variable contains a list of classes which will run as IU. If you don't believe me you can double check by just selecting out the RunAs property (the default table view won't show it) using the following:

```
PS C:\> $runas | Select Name, RunAs

Name              RunAs
----              -----
BrowserBroker Class   Interactive User
User Notification     Interactive User
...
```

On my machine I have around 200 classes installed that will run as IU. But that's not the end of the story, only a subset of these classes will actually be accessible from a sandbox such as Edge or cross-session. We need a way of filtering them down further. To filter we'll need to look at the associated security of the class registration, specifically the Launch and Access permissions. In order to launch the new object and get an instance of the class we'll need to be granted Launch Permission, then in order to access the object we get back we'll need to be granted Access Permissions. The class object exposes this as the LaunchPermission and AccessPermission properties respectively. However, these just contain a Security Descriptor Definition Language (SDDL) string representation of the security descriptor, which isn't easy to understand at the best of times. Fortunately I've made it easier, you can use the *Select-ComAccess* cmdlet to filter on classes which can be accessed from certain scenarios.

Let's first look at what objects we could access from the Edge content sandbox. First we need the access token of a sandboxed Edge process. The easiest way to get that is just to start Edge and open the token from one of the *MicrosoftEdgeCP* processes. Start Edge, then run the following to dump a list of the content processes.

```
PS C:\> Get-Process MicrosoftEdgeCP | Select Id, ProcessName

   Id ProcessName
   -- -----------
 8872 MicrosoftEdgeCP
 9156 MicrosoftEdgeCP
10040 MicrosoftEdgeCP
14856 MicrosoftEdgeCP
```

Just pick one of the PIDs, for this purpose it doesn't matter too much as all Edge CP's are more or less equivalent. Then pass the PID to the *-ProcessId* parameter for *Select-ComAccess* and pipe in the *$runas* variable we got from before.

```
PS C:\> $runas | Select-ComAccess -ProcessId 8872 | Select Name
```

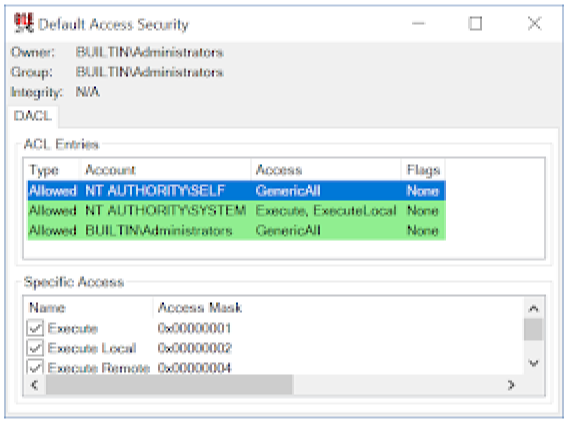
```
Name
----
PerAppRuntimeBroker
...
```

On my system, that reduces the count of classes from 200 to 9 classes, which is a pretty significant reduction. If I rerun this command with a normal UWP sandboxed process (such as the calculator) that rises to 45 classes. Still fewer than 200 but a significantly larger attack surface. The reason for the reduction is Edge content processes use Low Privilege AppContainer (LPAC) which heavily cuts down inadvertent attack surface.

What about cross-session? The distinction here is you'll be running as one unsandboxed user account and would like to attack the another user account. This is quite important for the security of COM objects, the default access security descriptor uses the special SELF SID which is replaced by the user account of the process hosting the COM server. Of course if the server is running as a different user in a different session the defaults won't grant access. You can see the default security descriptor using the following command:

```
Show-ComSecurityDescriptor -Default -ShowAccess
```

This command results in a GUI being displayed with the default access security descriptor. You see in this screenshot that the first entry grants access to the SELF SID.



Owner: BUILTIN\Administrators
Group: BUILTIN\Administrators
Integrity: N/A

ACL Entries

Type	Account	Access	Flags
Allowed	NT AUTHORITY\SELF	GenericAll	None
Allowed	NT AUTHORITY\SYSTEM	Execute, ExecuteLocal	None
Allowed	BUILTIN\Administrators	GenericAll	None

Specific Access

Name	Access Mask
✓ Execute	0x00000001
✓ Execute Local	0x00000002
✓ Execute Remote	0x00000004

To test for accessible COM classes we just need to tell the access checking code to replace the SELF SID with another SID we're not granted access to. You can do this by passing a SID to the *-Principal* parameter. The SID can be anything as long as it's not our user account or one of the groups we have in our access token. Try running the following command:

```
PS C:\> $runas | Select-ComAccess -Principal S-1-2-3-4 | Select Name

Name
----
BrowserBroker Class
...
```

On my system that leaves around 54 classes, still a reduction from 200 but better than nothing and still gives plenty of attack surface.

Inspecting COM Objects

I've only shown you how to find potential targets to look at for sandbox escape or cross-session attacks. But the class still needs some sort of way of elevating privileges, such as a method on an interface which would execute an arbitrary executable or similar. Let's quickly look at some of the functions in the PS module which can help you to find this functionality. We'll use the example of the *HxHelpPane* class I abused previously (and is now fixed as a cross-session attack in Project Zero Issue 1224, probably).

The first thing is just to get a reference to the class object for the *HxHelpPane* server class. We can get the class using the following command:

```
PS C:\> $cls = Get-ComClass -Name "AP Client HxHelpPaneServer Class"
```

The *$cls* variable should now be a reference to the class object. First thing to do is find out what interfaces the class supports. In order to access a COM object OOP you need a registered COM proxy. We can use the list of registered proxy interfaces to find what the object responds to. Again I have command to do just that, *Get-ComClassInterface*. Run the following command to get back a list of interface objects:

```
PS C:\> Get-ComClassInterface $cls | Select Name, Iid

Name           Iid
----          ---
IMarshal        00000003-0000-0000-c000-000000000046
IUnknown        00000000-0000-0000-c000-000000000046
IMultiQI        00000020-0000-0000-c000-000000000046
IClientSecurity  0000013d-0000-0000-c000-000000000046
IHxHelpPaneServer 8cec592c-07a1-11d9-b15e-000d56bfe6ee
```

Sometimes there's interesting interfaces on the factory object as well, you can get the list of interfaces for that by specifying the *-Factory* parameter to *Get-ComClassInterface*. Of the interfaces shown only *IHxHelpServer* is unique to this class, the rest are standard COM interfaces. That's not to say they won't have interesting behavior but it wouldn't be the first place I'd look for interesting methods.

The implementation of these interfaces are likely to be in the COM server binary, where is that? We can just inspect the DefaultServer property on the class object.

```
PS C:\> $cls.DefaultServer
C:\Windows\helppane.exe
```

We can now just break out IDA and go to town? Not so fast, it'd be useful to know exactly what we're dealing with before then. At this point I'd recommend at least using my tools NDR parsing code to extract how the interface is structured. You can do this by pass an interface object from G*et-ComClassInterface* or just normal *Get-ComInterface* into the *Get-ComProxy* command. Unfortunately if you do this you'll find a problem:

```
PS C:\> Get-ComInterface -Name IHxHelpPaneServer | Get-ComProxy
Exception: "Error while parsing NDR structures"
At OleViewDotNet.psm1:1587 char:17
+ [OleViewDotNet.COMProxyInterfaceInstance]::GetFromIID($In
+
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   + CategoryInfo          : NotSpecified: (:) []
   + FullyQualifiedErrorId : NdrParserException
```

This could be bug in my code, but there's a more likely reason. The proxy could be an auto-created proxy from a type library. We can check that using the following:

```
PS C:\> Get-ComInterface -Name IHxHelpPaneServer

Name              IID            HasProxy   HasTypeLib
----              ---            --------   ----------
IHxHelpPaneServer    8cec592c-07a1... True       True
```

We can see if the output that the interface has a registered type library, for an interface this likely means its proxy is auto-generated. Where's the type library? Again we can use another database command, *Get-ComTypeLib* and pass it the IID of the interface:

```
PS C:\> Get-ComTypeLib -Iid 8cec592c-07a1-11d9-b15e-000d56bfe6ee

TypelibId  : 8cec5860-07a1-11d9-b15e-000d56bfe6ee
Version    : 1.0
Name       : AP Client 1.0 Type Library
Win32Path  : C:\Windows\HelpPane.exe
Win64Path  : C:\Windows\HelpPane.exe
Locale     : 0
NativePath : C:\Windows\HelpPane.exe
```

Now you can use your favourite tool to decompile the type library to get back your interface information. You can also use the following command if you capture the type library information to the variable *$tlb:*

```
PS C:\> Get-ComTypeLibAssembly $tlb | Format-ComTypeLib
...
[Guid("8cec592c-07a1-11d9-b15e-000d56bfe6ee")]
interface IHxHelpPaneServer
{
  /* Methods */
  void DisplayTask(string bstrUrl);
  void DisplayContents(string bstrUrl);
  void DisplaySearchResults(string bstrSearchQuery);
  void Execute(string pcUrl);
}
```

You now know the likely names of functions which should aid you in looking them up in IDA or similar. That's the end of this quick tutorial, there's plenty more to discover in the PS module you'll just have to poke around at it and see. Happy hunting.