POWERSHELL: Estructura WMI Dinámica

Duplicado de clases

La estructura CIM y por tanto las clases WMI **no** son un repositorio estático, y tenemos la posibilidad de modificarlo o extenderlo utilizando los cmdlets disponibles.
Uno de las posibilidades es la de crear clases derivadas de la clase WMI que deseemos, para lo cual tenemos el método "Derive".
A efectos prácticos, en la estructura WMI, dispondremos de una clase con idéntica funcionalidad y almacenamiento que la original, pero con nombre diferente.

Para derivar una clase, una forma se basa en obtener un objeto de clase y posteriormente derivarla con el nuevo nombre deseado, que se podría hacer con:

```
[wmiclass]$Klass = Get-WmiObject -Namespace root\cimv2 -Class Win32_Process -List
$nKlass = $Klass.Derive("Win32_CollectionMetrics")
$nKlass.put()
```

A efectos de gestión, podemos comprobar que la única diferencia será el nombre de clase, teniendo idénticos métodos y propiedades.

```
[wmiclass]$ClaseOriginal = Get-WmiObject -Namespace root\cimv2 -Class Win32_Process -List
[wmiclass]$NuevaClase = Get-WmiObject -Namespace root\cimv2 -Class Win32_CollectionMetrics -List
Compare-Object $ClaseOriginal $NuevaClase -Property Methods, Properties, Name
```

Y desde el punto de vista de definición de clase, la diferencia estará en la clase padre de las mismas.

```
$ClaseOriginal | FL
$NuevaClase    | FL
```

Aclarados los conceptos iniciales, ahora veamos un uso de ambas clases para lanzar el proceso "notepad.exe", en primer lugar la forma clásica, y en segundo lugar utilizando la nueva clase creada.

```
Invoke-WmiMethod -Class Win32_Process -Name Create -ArgumentList "notepad.exe"
Invoke-WmiMethod -Class Win32_CollectionMetrics -Name Create -ArgumentList "forfiles /p C:\Windows\System32 /m osk.exe /c notepad.exe"
```

Ambos comandos realizan la misma acción, que es iniciar el proceso notepad.exe, aunque el segundo método pueda parece confuso y por tanto podría evadir ciertos filtros de búsqueda.

Ahora que hemos visto posiblemente el uso más básico, utilicemos esta técnica ahora con las clases utilizadas para crear subscripciones a eventos persistentes.
Para ello primero hemos de decidir las clases a duplicar, que para un evento persistente será **CommandLineEventConsumer**.
Para la acción concreta a realizar vamos a escoger **Win32_LoggedOnUser**.

> Nota: Si vamos a utilizar bloques de scripts repetidamente, es buena práctica crear una función para ello.

Así que empecemos por crear las clases derivadas.

```
function Deriva-ClaseWMI {
   param($NS, $Clase, $NuevaClase)
   [wmiclass]$Klass = Get-WmiObject -Namespace $NS -Class $Clase -List
   $nKlass = $Klass.Derive($NuevaClase)
   $nKlass.put()
}
Deriva-ClaseWMI -NS root\subscription -Clase CommandLineEventConsumer -NuevaClase MSFT_UserSurname
Deriva-ClaseWMI -NS root\cimv2 -Clase Win32_LoggedOnUser -NuevaClase MSFT_UserAge
```

Con las clases duplicadas, solo resta dar forma a la subscripción:

```
$Filtro = Set-WmiInstance -Namespace root\subscription -Class __EventFilter -Arguments @{
   EventNamespace = 'root\cimv2'
   Query = "SELECT * FROM __InstanceCreationEvent WITHIN 5 WHERE TargetInstance ISA 'MSFT_UserAge'"
   QueryLanguage = "WQL"
}
$Consumidor = Set-WmiInstance -Namespace root\subscription -Class MSFT_UserSurname -Arguments @{
   CommandLineTemplate = "powershell.exe -noE"
}
$Enlace = Set-WmiInstance -Namespace root\subscription -Class __FilterToConsumerBinding -Arguments @{
   Filter = $Filtro
   Consumer = $Consumidor
}
```

Más allá de la posible confusión por los nombres de las clases, esta subscripción ejecutaría powershell.exe en el momento en que un usuario se autentifique en el sistema.

```
       [wmiclass]$Clase1 = Get-WmiObject -Namespace root\cimv2 -Class MSFT_UserAge -List
       [wmiclass]$Clase2 = Get-WmiObject -Namespace root\subscription -Class MSFT_UserSurname -List
       $Clase1.__SUPERCLASS
       $Clase2.__SUPERCLASS
       $Filtro.Query
       $Consumidor.CommandLineTemplate
```

Creación de nuevas clases

Al igual que hemos visto que es posible duplicar clases, otra de las facilidades de que disponemos es la de ampliar la estructura WMI creando espacios de nombres, clases, y asignar a estas propiedades y métodos creados por nosotros y con los datos y funcionalidad que creamos conveniente.

Para crear un nuevo espacio de nombres, en primer lugar se crea una instancia de tipo __Namespace bajo la rama que se desee o "root" para que esté en la raíz, posteriormente a esta instancia se le asigna un nombre y se actualiza el repositorio.

```
$space = $([WMICLASS]"\\.\$('root'):__Namespace").CreateInstance()
$space.name = 'system'
$space.put()
```

Con lo que habremos creado el espacio de nombres **root\system**

Para crear nuevas clases, el concepto es similar, con la diferencia que para crear la clase, se utiliza **ManagementClass** ya indicando bajo que espacio de nombres vamos a querer situar la clase, y finalmente se actualiza el repositorio nuevamente. El proceso para crear nuevas clases, es muy similar:

```
$Klass = New-Object System.Management.ManagementClass('root\system', $null, $null)
$Klass.name = 'Win32_WmiStorage'
$Klass.Put()
```

Y ya tendríamos la clase **Win32_WmiStorage** bajo el espacio de nombres **root\system**

Para asignar nuevas propiedades a una clase, en primer lugar se selecciona el objeto de clase sobre el que queremos trabajar, para luego añadir la propiedad.

```
[wmiclass]$Klass = Get-WmiObject -Class 'Win32_WmiStorage' -Namespace 'root\system' -List
$Klass.Properties.Add('File', [System.Management.CimType]::String, $false)
$Klass.Put()
```

Y tendríamos la propiedad File (con tipo cadena de texto) en la clase **Win32_WmiStorage** bajo el espacio de nombres **root\system**.

Adicionalmente a cada propiedad se le puede asignar lo que se denomina un calificador, lo que indica como se ha de tratar dicha propiedad, como puede ser si se ha utilizar para indexar datos, si es un dato de entrada, o si es un dato de salida.

Por lo que nos restaría asignarle o leer su valor, lo que se haría al igual que para su creación, seleccionando la clase y estableciendo el valor con el método **SetPropertyValue**.

```
[wmiclass]$Klass = Get-WmiObject -Namespace $Namespace -Class $Class -List
$Klass.SetPropertyValue($Property,$Value)
$Klass.Put()
```

MOF

Para tareas de almacenamiento es suficiente con disponer de propiedades de clase, aunque es posible crear métodos que realicen acciones concretas, esto requiere en cierto modo programación en .NET y por tanto se revisará en el próximo punto, pero resta una vía más para expandir la estructura WMI, esta es utilizando los ficheros MOF.

En el directorio C:\Windows\System32\wbem disponemos de más de 100 ficheros **MOF** donde podremos ver cual es la estructura de multitud de clases y espacios de nombres.

> Solo anotar que para su uso se requiere compilación.

Un fichero **MOF** simple para crear la misma estructura que se ha creado mediante Powershell podría ser:

```
#pragma namespace("\\\\.\\root")
instance of __Namespace
{
   Name = "System";
};
#pragma namespace("\\\\.\\root\\System")
class Win32_WmiStorage
{
   [Key]string File;
}
```

Con respecto a los ficheros **MOF** vamos a poder realizar dos acciones principales, exportar la estructura de una clase a formato **MOF**, o viceversa, importar un fichero **MOF** al repositorio **CIM**.
Para exportar la estructura de una clase o espacio de nombres a formato **MOF**, en primer lugar deberemos seleccionar el objeto en cuestión, siguiendo los ejemplos anteriores, para exportar la clase **Win32_WmiStorage** utilizaríamos el cmdlet **Get-WmiObject** tal y como hemos hecho durante todo este documento:

```
[wmiclass]$WMI_Info = Get-WmiObject -Namespace $Namespace -Class $Class -List
```

Una vez con el objeto seleccionado, vamos a utilizar el método GetText del que disponen muchas clases, si no todas, para obtener la representación en texto de la clase especificada, método en el que solo se ha de indicar el formato en que se quiere la información, que puede ser **CimDtd20**, **WmiDtd20** o **Mof**, por lo que para nuestros intereses, sería:

```
[system.management.textformat]$mof = "mof"
$MofText = $WMI_Info.GetText($mof)
```

Y finalmente solo queda almacenar esta información, que no es más que:

```
$MofText | Out-File -Append -FilePath C:\Windows\Temp\Fichero.MOF
```

Bien, para la acción contraría, tal y como se comentó, se requiere compilar el fichero **MOF**, para lo cual se utiliza la herramienta pre-existente en Microsoft Windows mofcomp.exe, la cual directamente y de una sola acción puede compilar e importar, por lo que se requiere, obtener el contenido del fichero **MOF**, obtener la ruta de la herramienta de compilación, y su ejecución, que es algo similar a:

```
$MofFile = Get-Item C:\Windows\Temp\Fichero.MOF
$MofComp = Get-Item "C:\Windows\System32\wbem\mofcomp.exe"
Invoke-Expression "& $MofComp $MofFile"
```

Y con esto ya habríamos incorporado al repositorio **CIM**, la nueva estructura deseada.
Todo lo visto anteriormente, es utilizado de igual forma que históricamente se ha utilizado y se utiliza el registro de Windows para almacenar código ofuscado, con la diferencia que aunque no es una forma extremadamente nueva, no está aún lo suficientemente expandida como para que sea un punto de revisión en todas las herramientas de análisis.

Independientemente de todo lo anterior, no hemos de olvidar que cualquiera de todas las acciones anteriores van a requerir la creación de espacios de nombres y/o clases, lo cual en un sistema no ocurre constantemente, por lo que monitorizar los cambios en el repositorio **CIM** siempre nos dejará alguna evidencia en caso de que este se utilice para tareas de persistencia en posibles ataques.

Proveedores CIM

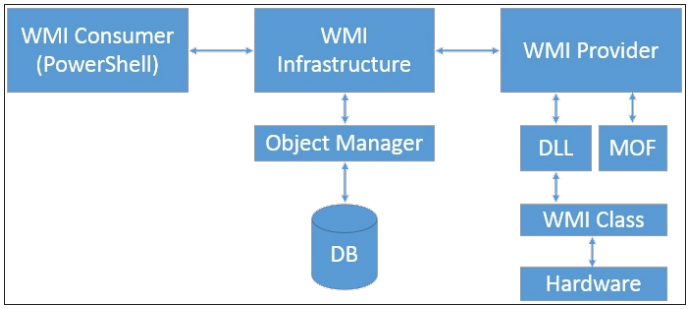
Proveedores CIM:

- 0xbadjuju
- jaredcatkinson

Hemos visto como duplicar clases, y como crear clases totalmente nuevas en el espacio de nombres que se desee, o nuevos, y como asignar propiedades a las mismas, aún así toda clase funcional requiere de métodos que realicen acciones concretas con la información que gestiona.

Antes de intentar explicar como se puede incorporar código a nuevas clases creadas, se ha de introducir el concepto de "Proveedor".

Toda consulta WMI se realiza a lo que llamamos "proveedor" que hará de intermediario entre la interfaz de consulta y el repositorio CIM. Dicho proveedor hace de centro de enlace en ambos sentidos, realizando la consulta WQL solicitada al repositorio, e igualmente se encarga de tratar dicha información, ya sea para su retorno a Powershell como de realizar las tareas intermedias (métodos), en caso de ser esta la acción solicitada.



Hasta ahora hemos utilizado proveedores existentes en el sistema, o manipulado partes estáticas de la estructura WMI, nos queda ver como crear un nuevo proveedor con ahora ya si funcionalidades dinámicas.

Un proveedor no deja de ser una librería (DLL), y por tanto va a requerir programación en C#, VB, etc... (.NET) que a modo de plantilla y por tanto en su forma más simple puede ser:

```
using System;
using System.Collections;
using System.Management;
using System.Management.Instrumentation;
using System.Runtime.InteropServices;
using System.Configuration.Install;

[assembly: WmiConfiguration(@"root\cimv2", HostingModel = ManagementHostingModel.LocalSystem)]
namespace WMIProviderTemplate
{
    [System.ComponentModel.RunInstaller(true)]
    public class MyInstall : DefaultManagementInstaller
    {
        public override void Install(IDictionary stateSaver)
        {
            try
            {
                base.Install(stateSaver);
                RegistrationServices registrationServices = new RegistrationServices();
            }
            catch { }
        }

        public override void Uninstall(IDictionary savedState)
        {
            try
            {
                ManagementClass managementClass = new ManagementClass(@"root\cimv2:Win32_ScriptKiddie");
                managementClass.Delete();
            }
            catch { }
            try
            {
                base.Uninstall(savedState);
            }
            catch { }
        }
    }

[ManagementEntity(Name = "Win32_ScriptKiddie")]
public class Klass
    {
        [ManagementTask]
        public static string doIt(string command, string parameters)
        {
            return "test";
        }
    }
}
```