At a glance:
- What is Windows PowerShell?
- How GPOs were managed previously
- Migrating scripts to Windows PowerShell

Simplify Group Policy administration with Windows PowerShell

Thorbjörn Sjövold

Microsoft Group Policy technology didn't catch on overnight – it was somewhat difficult to understand and required that you adopt Active Directory, a strange, new service that looked nothing like the Account/Resource domains that were the standard at the time. Today, Group Policy is the management backbone of nearly every organisation with a Windows infrastructure. I have a feeling this is exactly what will happen with Windows PowerShell™, the latest management technology from Microsoft. In fact, Windows PowerShell is likely to make your job as a Group Policy administrator significantly easier.

In this article, I show how the Microsoft Group Policy Management Console (GPMC) APIs written for Windows Scripting Host languages like VBScript (and COM-based scripting languages in general) can be consumed directly from Windows PowerShell to simplify the way you manage Group Policy in your environment.

Scripting Group Policy tasks

When Microsoft released GPMC a few years ago, Group Policy administrators suddenly had a number of handy features at their disposal. In particular, the Group-Policy-focused MMC snap-in represented a huge step forward for Group Policy management, especially compared to Active Directory Users

and Computers. Moreover, there was a brand new API that let administrators use COM-based languages such as VBScript to perform Group Policy administration of tasks, such as backing up and restoring Group Policy Objects (GPOs), migrating GPOs between domains, configuring security settings on GPOs and links, and reporting.

> The vast amount of .NET code being produced today can be used directly from within Windows PowerShell

Unfortunately, GPMC did not provide the ability to edit the actual configured settings inside the GPOs. In other words, you could perform operations on the GPO container, such as reading GPO versions, reading modify dates, creating brand new GPOs, backing up and restoring/importing GPOs from different domains, and so forth, but you couldn't programmatically add or change the content of the GPO, for example adding a new redirected folder or a new software installation. What you generally did instead was create the GPO and configure all settings manually using the Group Policy Object Editor, then back it up and import it into a test environment. When everything was verified and working properly, you would import it into the live environment. Despite the missing feature, the use of scripts instead of manual interaction with the GPMC API resulted in a tremendous amount of time, effort and error saved in day-to-day Group Policy administration.

The next level

How is Windows PowerShell different from scripting languages like VBScript? For starters, Windows PowerShell is a shell and, at least for our purposes, you can think of a shell as a command-line interpreter. Though VBScript can be run from the command line, a VBScript file cannot be run line by line. A Windows PowerShell script, in contrast, can be created on the fly as a series of individual commands. In addition, Windows PowerShell has functions that work much like subroutines in VBScript, and that can be created in real time at the Windows PowerShell command prompt.

Even better, Windows PowerShell is built on the Microsoft .NET Framework, while VBScript relies on older COM technology. This means that the vast amount of .NET code being produced today can be used directly from within Windows PowerShell.

What it comes down to is that with Windows PowerShell, you get full scripting support and interactive mode, all in one package. The examples I provide here will all be command-line input so you can type as you read; however, they'll work equally well if you put them in a Windows PowerShell script file and run it.

Recreate old scripts using Windows PowerShell

The last thing you want to do when you start with a new technology is toss out all your previous work. There are three approaches you can use to access COM objects from the GPMC API, or basically to reuse any old VBScript out there. You can select one of these three options:

- Create a Windows PowerShell cmdlet using a programming language like C# or managed C++.
- Use Windows PowerShell to access the ScriptControl in MSScript.ocx to wrap old scripts.
- Wrap the COM calls in reusable Windows PowerShell functions or else call the COM objects directly.

I'm going to focus mainly on the third option, but let's have a quick look at all the options first.

Creating a Windows PowerShell Cmdlet

Microsoft included a large number of cmdlets with Windows PowerShell, allowing you to copy files, format output, retrieve the date and time, and so on, but you can create your own cmdlets as well. The process is fully documented at www.microsoft.com/uk/technetmagazine/cmdlet. Here are the steps:

- Create a class library DLL in a .NET programming language such as C#.

- Create a new class and inherit from the base class cmdlet.
- Set attributes that determine the name, usage, input parameters, and so forth, and add your code.

Because Windows PowerShell is built on the .NET Framework, any types, such as a string, object, and so forth, that are being returned or passed as parameters are exactly the same in the code as in Windows PowerShell; no special type conversion is needed.

The real power of this solution is that you have a complete programming language at your disposal.

Wrapping old scripts using the ScriptControl object in MSScript.ocx

Obviously, you need the VBScript engine to run a VBScript file. What is not so obvious is that this engine is a COM object and, since you can use COM objects from Windows PowerShell, you can invoke the VBScript engine. Here's how this could look:

```
$scriptControl = New-Object -ComObject ScriptControl
$scriptControl.Language = 'VBScript'
$scriptControl.AddCode(
    'Function ShowMessage(messageToDisplay)
    MsgBox messageToDisplay
    End Function')
$scriptControl.ExecuteStatement('ShowMessage
    "Hello World"')
```

If you enter this code in the Windows PowerShell Command Line Interface (CLI), the VBScript function ShowMessage is called and executed with a parameter, resulting in a Message Box displayed with the text Hello World.

Now some of you might think, "Great! I've mastered the art of using COM from Windows PowerShell and I can stop reading this article and start filling the ScriptControl object with my collection of old GPMC scripts." Unfortunately, that's not the case. This technique quickly becomes very complex and tough to debug as soon as the scripts get larger.

Wrapping COM objects

So the best option is the third: wrapping the COM calls in reusable Windows PowerShell functions, which lets you consume the COM objects in the GPMC API. The line below shows how to create a .NET object directly in Windows PowerShell. In this case, it's a FileInfo object that can be used to get the size of the file:

```
$netObject = New-Object System.IO.FileInfo(
"C:\boot.ini") # Create an instance of FileInfo
               # representing c:\boot.ini
```

Note that # is used in Windows PowerShell for inline comments. Using this newly instantiated FileInfo object, you can easily get the size of boot.ini by simply typing the following code:

```
$netObject.Length # Display the size in bytes of the
                  # file in the command line interface
```

Wait, weren't we supposed to talk about COM objects and VBScript conversion? Yes, but look at the following command:

```
$comFileSystemObject = New-Object -ComObject
Scripting.FileSystemObject
```

You'll notice that the syntax is basically the same as I used previously to create native objects from the .NET Framework, with two differences. First, I added the –ComObject switch that points Windows PowerShell toward the COM world instead of the .NET world. Second, I use a COM ProgID instead

Figure 1 Custom functions in the download

Function Name	Description
BackupAllGpos	Backs up all GPOs in a domain
BackupGpo	Backs up a single GPO
RestoreAllGpos	Restores all GPOs from a backup to a domain
RestoreGpo	Restores a single GPO from a backup
GetAllBackedUpGpos	Retrieves the latest version of GPO backups from a given path
CopyGpo	Copies the settings in a GPO to another GPO
CreateGpo	Creates a new empty GPO
DeleteGpo	Deletes a GPO
FindDisabledGpos	Returns all GPOs where both the user and computer part are disabled
FindUnlinkedGpos	Returns all GPOs that have no links
CreateReportForGpo	Creates an XML report for a single GPO in a domain
CreateReportForAllGpos	Creates a separate XML report for each GPO in a domain
GetGpoByNameOrID	Finds a GPO by its display name or GPO ID
GetBackupByNameOrId	Finds a GPO backup by its display name or GPO ID
GetAllGposInDomain	Returns all GPOs in a domain

Wrap the COM calls in reusable Windows PowerShell functions

of a .NET constructor, in this case Scripting.FileSystemObject. The ProgID is the same name you've always used. In VBScript, the equivalent would be:

```
Set comFileSystemObject = CreateObject(
    "Scripting.FileSystemObject")
```

To get the file size using VBScript, add the line above to a file, together with the following code:

```
Set comFileObject = comFileSystemObject.GetFile(
    "C:\Boot.ini")
WScript.Echo comFileObject.Size
```

Then run it using Cscript.exe, for example. In Windows PowerShell, you would do it like this (directly from the Windows PowerShell command line if you like):

```
$comFileObject = $comFileSystemObject.GetFile(
    "C:\boot.ini")
$comFileObject.Size
```

Of course, to convert a VBScript that reads the size of a file, I could have used the Windows PowerShell cmdlet that manages objects in drives, but I wanted to show you how easy it is to access COM from Windows PowerShell. Note that though I tell Windows PowerShell to create a COM object, the object that actually gets created, $comFileSystemObject here, is a .NET object that wraps the COM object and exposes its interface. In the scope of this article, however, that doesn't really matter.

Windows PowerShell in action

Now that you've seen how to access COM from Windows PowerShell, let's focus on Group Policy. The examples here will show short code snippets to give you an idea of how to use the GPMC APIs from Windows PowerShell, but you'll find a full set of Windows PowerShell functions to manage Group Policy in the code download related to this article, online at technetmagazine.com/code07.aspx. **Figure 1** lists the functions included in the download.

As you read this section, feel free to start the Windows PowerShell command line and type in the commands. Remember that some commands are dependent on previous commands, however. In other words, some of the objects created initially will be used later on, so stay in the same Windows PowerShell session. If you close the session, you'll have to start again from the beginning, retyping all the commands.

So, let's create a new GPO using Windows PowerShell. The Group Policy team at Microsoft included a number of fully working VBScript samples with GPMC that you can take advantage of to speed things up. They're in the %ProgramFiles%\GPMC\Scripts directory, where you'll also find a file called gpmc.chm that contains the GPMC API documentation. Let's have a look at the CreateGPO.wsf script and dissect it to see what makes it tick.

Near the top you'll see this line:

```
Dim GPM
Set GPM = CreateObject("GPMgmt.GPM")
```

This is basically the starting point of any Group Policy management session or script because it instantiates the GPMgmt.GPM class that allows access to most of the GPMC functionality. Let's go ahead and do this from Windows PowerShell instead:

```
$gpm = New-Object -ComObject GPMgmt.GPM
```

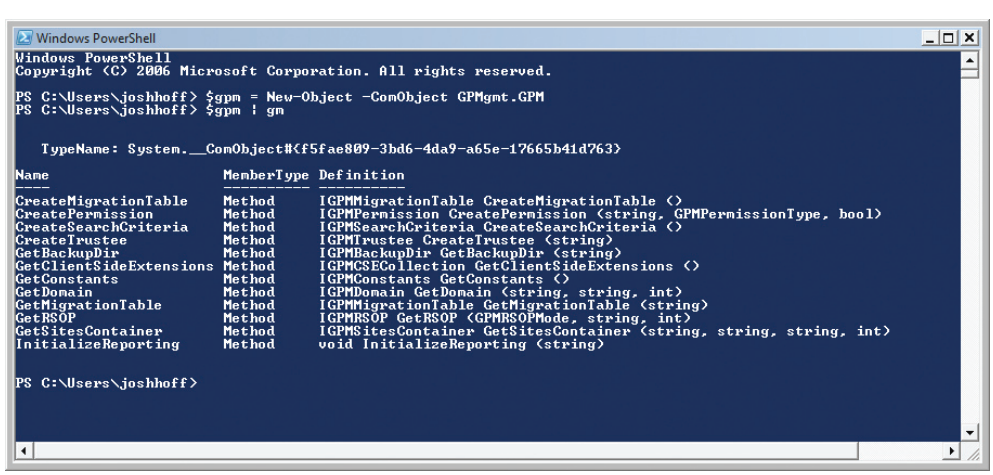

Figure 2 **Get-Member output**

Now that you have the starting point for Group Policy management, the next step is to figure out what you can do with it. Normally, you'd turn to the documentation for this type of information, but Windows PowerShell has a really cool feature. If you type the following line, you get the output shown in **Figure 2**:

```
$gpm | gm
```

Pretty cool if you ask me. Note how the Get-Member (or gm) cmdlet lets you see the properties and methods the object supports right from the command line. Of course it's not the same as reading the documentation, but it makes it easy to use objects you're already familiar with when you don't remember the exact number of parameters, the exact name, and so forth. One important point to note is that when you look at the GPMC documentation node listings, it looks like the GPM object and all the other classes are prefixed with the letter I; this is due to the internal workings of COM and doesn't really concern us here; it is intended for C++ programmers writing native COM code and denotes the difference between an interface and the class that implements it. Also note that when using the GPMC APIs, there is only one object you need created this way, and that is GPMgmt.GPM; all other objects are created using methods that start with this GPM object.

Let's continue now with the creation of a new GPO.

Figure 3 illustrates how simple it is to create a GPO. Note that I've left out some code, including error handling (for example, what will happen if you are not allowed to create GPOs), and I've hardcoded a domain name, but you should get the idea.

Now that you know how to create a GPO, let's open an existing one instead. You still have the reference to the domain, $gpmDomain, so type in the following:

```
$gpmExistingGpo = $gpmDomain.GetGPO(
  "{31B2F340-016D-11D2-945F-00C04FB984F9}")
# Open an existing GPO based on its GUID,
# in this case the Default Domain Policy.
$gpmExistingGpo.DisplayName
# Show the display name of the GPO, it
# should say Default Domain Policy
$gpmExistingGpo.GenerateReportToFile($gpmConstants.
ReportHTML, ".\DefaultDomainPolicyReport.html"
```

This gives you a full HTML report of the settings in the Default Domain Policy, but you can obviously use any of the methods and properties, like ModificationTime, which tells you when the GPO was last modified, to figure out when any of the settings in the GPO changed.

This is extremely useful. Most likely you've been in a situation where the phones start ringing like crazy with users complaining that their computers are acting weird. You suspect this is related to a changed, added or deleted GPO setting, but you don't have a clue which GPO to look in. Windows PowerShell to the rescue! If you enter the script shown in **Figure 4** at the Windows PowerShell command line, you get all GPOs that have been changed in the last 24 hours.

Note the –ge operator, which means greater than or equal to. It might look strange if you're used to the < and > operators in other scripting or programming languages. But those operators are used for redirection, for example to redirect the output to file, and thus could not be used as comparison operators in Windows PowerShell.

Wrapping up

The code in **Figure 5** lists the full script for copying the settings in one GPO to another GPO. You should now have a good idea of how you can use this new technology

Figure 3 Create a GPO

```
$gpmConstants = $gpm.GetConstants()
# This is the GPMC way to retrieve all
# constants
$gpmDomain =$gpm.GetDomain("Mydomain.local", "", $gpmConstants.UseAnyDC)
# Connects to the domain where the GPO should
# be created, replace Mydomain.local with the
# name of the domain to connect to.
$gpmNewGpo = $gpmDomain.CreateGPO()
# Create the GPO
$gpmNewGpo.DisplayName = "My New Windows PowerShell GPO"
# Set the name of the GPO
```

Figure 4 Discovering modified GPOs

```
$gpmSearchCriteria = $gpm.CreateSearchCriteria()
# We want all GPOs so no search criteria will be specified
$gpmAllGpos = $gpmDomain.SearchGPOs($gpmSearchCriteria)
# Find all GPOs in the domain
foreach ($gpmGpo in $gpmAllGpos)
{
if ($gpmGpo.ModificationTime -ge (get-date).AddDays(-1)) {$gpmGpo.DisplayName}
# Check if the GPO has been modified less than 24 hours from now
}
```

Figure 5 **Copy the settings in one GPO to another GPO**

```
#####################################################################
# Function  : CopyGpo
# Description: Copies the settings in a GPO to another GPO
# Parameters : $sourceGpo     - The GPO name or GPO ID of the GPO to copy
#            : $sourceDomain   - The dns name, such as microsoft.com, of the domain where the original GPO is located
#            : $targetGpo      - The GPO name of the GPO to add
#            : $targetDomain   - The dns name, such as microsoft.com, of the domain where the copy should be put
#            : $migrationTable - The path to an optional Migration table to use when copying the GPO
# Returns   : N/A
# Dependencies: Uses GetGpoByNameOrID, found in article download
#####################################################################
function CopyGpo(
 [string] $sourceGpo=$(throw '$sourceGpo is required'),
 [string] $sourceDomain=$(throw '$sourceDomain is required'),
 [string] $targetGpo=$(throw '$targetGpo is required'),
 [string] $targetDomain=$(throw '$targetDomain is required'),
 [string] $migrationTable=$(""),
 [switch] $copyAcl)
{

 $gpm = New-Object -ComObject GPMgmt.GPM # Create the GPMC Main object
 $gpmConstants = $gpm.GetConstants() # Load the GPMC constants
 $gpmSourceDomain = $gpm.GetDomain($sourceDomain, "", $gpmConstants.UseAnyDC) # Connect to the domain passed
                                                                               # using any DC
 $gpmSourceGpo = GetGpoByNameOrID $sourceGpo $gpmSourceDomain
 # Handle situations where no or multiple GPOs was found
 switch ($gpmSourceGpo.Count)
 {
   {$_ -eq 0} {throw 'No GPO named $gpoName found'; return}
   {$_ -gt 1} {throw 'More than one GPO named $gpoName found'; return}
 }
 if ($migrationTable)
 {
   $gpmMigrationTable = $gpm.GetMigrationTable($migrationTable)
 }

 $gpmTargetDomain = $gpm.GetDomain($targetDomain, "", $gpmConstants.UseAnyDC) # Connect to the domain passed
                                                                               # using any DC

 $copyFlags = 0
 if ($copyAcl)
 {
   $copyFlags = Constants.ProcessSecurity
 }
 $gpmResult = $gpmSourceGpo.CopyTo($copyFlags, $gpmTargetDomain, $targetGpo)
 [void] $gpmResult.OverallStatus

}
```

with Group Policy and how you can reuse any COM object or VBScript code that consumes a COM object.

Windows PowerShell will be, just as Group Policy already is, a natural part of any Windows management environment. But there are millions of lines of VBScript out there that will have to be migrated or maintained, and hopefully this tutorial will help.

There are a number of sources you can use to enhance your Group Policy administration and other areas where you've previously used VBScript, including the Windows PowerShell functions in the download, and a great VBScript-to-Windows PowerShell conversion guide on the TechNet Web site that provides hints on how to do common tasks in Windows PowerShell when you know the equivalent in VBScript. You'll find it at microsoft.com/technet/scriptcenter/topics/winpsh/convert.

In addition, the GPMC API is fully documented; you can download the information from the Group Policy site at microsoft.com/grouppolicy.

Last but not least, if you haven't already installed Windows PowerShell, what are you waiting for? Download it from microsoft.com/powershell. Have fun. ■

Thorbjörn Sjövold *is the CTO and founder of Special Operations Software (www.specopssoft.com), a provider of Group Policy based Systems Management and Security extension products. Reach him at thorbjorn.sjovold@specopssoft.com.*