We've discussed how to perform many of our day-to-day operations using the GPMC. But for real efficiency, consider automating day-to-day operations with scripts.

In this bonus chapter, we'll spend some time discussing how to use Windows PowerShell to document, manage, and interact with Group Policy. PowerShell provides a script interface and an interactive shell that makes day-to-day tasks fun (well, almost fun).

There are two ways administrators can use PowerShell with regard to Group Policy. One way is to use PowerShell to deploy PowerShell scripts to client machines. We described how to do that in detail in Chapter 12, “Finishing Touches with Group Policy: Scripts, Internet Explorer, Hardware Control, and Printer Deployment.”

In this chapter, we're taking a different PowerShell journey. We're going to leverage PowerShell to do “Group Policy stuff.”

A cmdlet (pronounced “command-let”) is like PowerShell commands in a prepackaged snack pack.

Using PowerShell to Do More with Group Policy

Before we get into heavy lifting with PowerShell, we'll cover some basics to get you up and running. First, we'll briefly discuss installing PowerShell. Then, we'll dive into doing some cool stuff with Group Policy, including automating our documentation and manipulating our GPOs. Finally, we'll deal with the sticky business of importing GPOs so that we can utilize them in a test lab before we import them into production.

This chapter was written by Jeffery Hicks, PowerShell MVP, with Jeremy Moskowitz, Group Policy MVP.
Everything we are going to be discussing in this chapter is based on PowerShell v3, which you get automatically with Windows 8 and Windows Server 2012. The stuff we talk about here will also work on Windows 7, Windows Server 2008, and Windows Server 2008 R2, although you’ll have to download PowerShell v3 for Windows 7 and Windows Server 2008 platforms. Almost everything we’re going to show you will also work with PowerShell v2, which ships by default with Windows 7 and Windows Server 2008. You can download PowerShell v2 for other Windows platforms, XP and later, from Microsoft. We’re also going to assume you want to manage Group Policy from the comfort of your cubicle without resorting to remote desktop sessions and the like.

For the most part, everything you need to accomplish can be done with any Active Directory domain flavor. We’ve tested Windows 2003 and later. In the book, you mainly use Windows 7 or 8 as your management machine, and now you’ll continue to use Windows 7 or 8 as your “PowerShell management machine.” We’ll explain how to set up your machine for this job in a page or so.

There are a few instances where we want to take advantage of a little something extra: add-ons are available that enable Active Directory management using PowerShell. Those goodies are wrapped inside the Active Directory cmdlets from Microsoft.

To use these new Active Directory cmdlets, you will need one of two things:

- A Windows Server 2008 R2 or later domain controller.
- If you only have a Windows Server 2008 (or 2003) domain, you can download and install the free Active Directory Management Gateway Service (ADMGS) from Microsoft at: http://tinyurl.com/ps423h

You can find a helpful blog entry on getting this all set up at:

With the ADMGS, you’ll still need a Windows 7 or later client to “ask” the ADMGS machines about Active Directory, when needed.

Again, the only reason you need a Windows Server 2008 or Windows Server 2008 R2 domain controller (or the ADMGS) is if you plan on using the Microsoft Active Directory cmdlets to work with objects like users, groups, and organizational units.

Otherwise, the “built-in stuff” in the box with Windows 8 will handle all the Group Policy stuff perfectly well—no additional installs required.

Preparing for Your PowerShell Experience

We recommend you continue to use your Windows 8 machine as your PowerShell management machine. That’s because it has everything you’ll need to be successful—immediately. You could also use a Windows 7 or Windows Server 2008 R2 machine—because the “guts” of Windows 8, Windows 7, and Windows Server 2008 R2 are all equivalent, and they can run the same PowerShell version and have the built-in stuff we need to manage GPOs.

For this chapter, we’ll assume you’ll use Windows 8 as your GPMC and your PowerShell management machine.
Running PowerShell for the First Time

Before we proceed, let’s make sure you can launch PowerShell. To run PowerShell on Windows 7, you have two options. When PowerShell is installed, the system path is added, so you can simply go to Start ➤ Run ➤ PowerShell.exe, or you can use the shortcut in the Programs menu. Specifics may vary by operating system. Or you can choose Start ➤ All Programs ➤ Accessories ➤ Windows PowerShell ➤ Windows PowerShell. If you are running a 64-bit operating system, you will most likely see options for 32- and 64-bit. For now, select the version that corresponds to your architecture.

Windows 8 is a little trickier with the new start interface. The easiest way to launch PowerShell is to press the Windows key and then start typing PowerShell.exe in the search box. We then recommend right-clicking the tile and pinning it to the Start Menu and the Taskbar. This way you can launch PowerShell from either location.

You should see a PowerShell window like the one shown in Figure BC1.1.

PREPARING TO RUN OUR SCRIPTS

Finally, there is one last thing to do. By default, as a security feature, PowerShell will not run any scripts. You can use PowerShell as a command-line interface (CLI) to run interactive commands, but it stops short of running scripts.

That’s right. It’s like having a Ferrari you can’t drive unless you have a special set of keys cut. And that’s what you’re going to do with this next command. You’ll be able to drive the Ferrari after you “cut the keys” by running a cmdlet called Set-ExecutionPolicy. PowerShell supports six modes in 2.0:

Unrestricted All scripts will run.
RemoteSigned If a script is downloaded from the Internet or from any site that is considered an Internet Zone, it must be digitally signed with a trusted code signing certificate.
AllSigned All scripts must be digitally signed with a trusted code signing certificate.
Restricted No scripts are allowed (this is the default).
Bypass  Nothing is blocked and there are no warnings or prompts.

Undefined  This mode removes the execution policy from the current scope as long as it
wasn’t defined by a Group Policy.

To change the mode, you need to run a simple command. We recommend setting the
mode to at least RemoteSigned. You do that with this command in an elevated PowerShell
session:

```powershell
PS C:\> Set-ExecutionPolicy RemoteSigned
```

We’re choosing RemoteSigned because it ensures that we can’t inadvertently run scripts
downloaded from the Internet. (We all know where that could lead.) We’ll be writing and
running our own scripts, so this is the perfect level of protection.

---

You can download the chapter’s scripts from us at www.GPanswers.com in
the “Book’s Resources” section.

---

Getting Started with PowerShell

Of course, you don’t want to, and shouldn’t need to, log onto a Domain Controller to man-
age your network. Instead, you will want to use your Windows 8 or Windows 7 computer.
All the examples in this chapter will be done from a Windows 8 desktop. There are a few
steps you need to take to get ready.

Windows 8 Remote Server Administration Tools

The first step is to download and install the Remote Server Administration Tools (RSAT)
for Windows 8. You did that way back in Chapter 1, “Group Policy Essentials.”

However, let’s review in case you’ve jumped right ahead here.

If you’re using Windows 8, install the RSAT for Windows 8, which (at last check) was
here: http://tinyurl.com/9dz427g.

If you’re using Windows 7, then install Windows 7’s RSAT from (at last check) here:
http://tinyurl.com/klycep.

Open Control Panel and then the Programs applet. Click “Turn Windows Features On
or Off.” Scroll down to the Remote Server Administration Tools and expand the list.

When you install RSAT on Windows 8, everything should be ready to go for what we need
for this chapter. But let’s verify everything is there. Expand Feature Administration Tools
and check Group Policy Management Tools. While you’re at it, expand Role Administration
Tools and then AD and LDS Tools, in case you may also want to enable the Active Directory
Module for PowerShell. Although we don’t need it for our work in this chapter, you probably
will be using it for other tasks. Of course, enable support for anything else you need to man-
age. You can see my result in Figure BC1.2.
**Adding the Group Policy Module**

Before you can use the Group Policy cmdlets in PowerShell v2, you must import the module:

```
PS C:\> import-module GroupPolicy
```

PowerShell v3 will autoload modules. What goodies do we get with this module? Let’s ask PowerShell:

```
PS C:\> get-command -module grouppolicy | select name | sort name
```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup-GPO</td>
</tr>
<tr>
<td>Copy-GPO</td>
</tr>
<tr>
<td>Get-GPInheritance</td>
</tr>
<tr>
<td>Get-GPO</td>
</tr>
<tr>
<td>Get-GPReport</td>
</tr>
<tr>
<td>Get-GPPermission</td>
</tr>
<tr>
<td>Get-GPPermissions</td>
</tr>
<tr>
<td>Get-GPPrefRegistryValue</td>
</tr>
</tbody>
</table>

---

**Figure BC1.2**  Turn Windows features on or off.
Get-GPRegistryValue
Get-GPResultantSetOfPolicy
Get-GPStarterGPO
Import-GPO
Invoke-GPUpdate
New-GPLink
New-GPO
New-GPStarterGPO
Remove-GPLink
Remove-GPO
Remove-GPPrefRegistryValue
Remove-GPRegistryValue
Rename-GPO
Restore-GPO
Set-GPIheritance
Set-GPLink
Set-GPPermission
Set-GPPermissions
Set-GPPrefRegistryValue
Set-GPRegistryValue

Cmdlet names follow a verb–dash–singular noun naming convention, which makes it pretty easy to figure out what to use. If you want more information, simply ask PowerShell for help, as seen in Figure BC1.3.

**FIGURE BC1.3** Getting cmdlet help in PowerShell
If you want even more help information, ask PowerShell for full help like this:

```powershell
PS C:\> help get-gpo -full
```

At the end of the help output, there will be a number of examples on how to use the cmdlet.

We won’t go through every single cmdlet you see listed in the GroupPolicy module; instead, we’ll focus on some common Group Policy management tasks.

**Documenting Your Group Policy World with PowerShell**

One of the most complicated and annoying parts of managing GPOs is the need to document them. Without clear documentation, troubleshooting can be quite complicated. In this next section, we’ll show you how easy it is to create useful documentation with the Group Policy cmdlets and PowerShell. You’ll be amazed at how much you can accomplish with having to write a single script.

**Listing GPOs**

Let’s start by listing all the GPOs within the domain. And in that list, we will, of course, retrieve some specific information about the GPO. We’ll get things like the name, where it’s linked in Active Directory, creation time, modification time, and much, much more.

```powershell
PS C:\> get-gpo -all -domain mycompany.local
```

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>Network Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainName</td>
<td>jdhlab.local</td>
</tr>
<tr>
<td>Owner</td>
<td>JDHLAB\Domain Admins</td>
</tr>
<tr>
<td>Id</td>
<td>019e061a-b032-48e7-a6ae-f0a0f37f809a</td>
</tr>
<tr>
<td>GpoStatus</td>
<td>AllSettingsEnabled</td>
</tr>
<tr>
<td>Description</td>
<td>IP and network related settings for all users</td>
</tr>
<tr>
<td>CreationTime</td>
<td>9/13/2010 3:17:41 PM</td>
</tr>
<tr>
<td>ModificationTime</td>
<td>9/13/2010 3:17:42 PM</td>
</tr>
<tr>
<td>UserVersion</td>
<td>AD Version: 0, SysVol Version: 0</td>
</tr>
<tr>
<td>ComputerVersion</td>
<td>AD Version: 0, SysVol Version: 0</td>
</tr>
<tr>
<td>WmiFilter</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>HR Desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainName</td>
<td>jdhlab.local</td>
</tr>
<tr>
<td>Owner</td>
<td>JDHLAB\Domain Admins</td>
</tr>
<tr>
<td>Id</td>
<td>07126f7c-fcc5-48ec-90dd-50ef5725708f</td>
</tr>
<tr>
<td>GpoStatus</td>
<td>UserSettingsDisabled</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>CreationTime</td>
<td>9/2/2010 4:55:11 PM</td>
</tr>
</tbody>
</table>
UserVersion : AD Version: 0, SysVol Version: 0
WmiFilter :

DisplayName : Win7 Special
DomainName : jdhlab.local
Owner : JDHLAB\Domain Admins
Id : 16b56faf-0746-4197-8860-c223b593e5af
GpoStatus : UserSettingsDisabled
Description :
CreationTime : 7/12/2010 4:05:38 PM
ModificationTime : 9/6/2010 8:47:10 AM
UserVersion : AD Version: 0, SysVol Version: 0
WmiFilter : Windows 7
DisplayName : Sales Desktop
DomainName : jdhlab.local
Owner : JDHLAB\Domain Admins
Id : 2c92da74-d0de-4870-92ff-a1dd7a70fc95
GpoStatus : AllSettingsEnabled
Description :
CreationTime : 9/2/2010 4:54:59 PM
ModificationTime : 9/14/2010 10:13:16 AM
ComputerVersion : AD Version: 1, SysVol Version: 1
WmiFilter :

DisplayName : Default Domain Policy
DomainName : jdhlab.local
Owner : JDHLAB\Domain Admins
Id : 31b2f340-016d-11d2-945f-00c04fb984f9
GpoStatus : AllSettingsEnabled
Description :
CreationTime : 1/24/2010 3:02:46 PM
ModificationTime : 6/30/2011 4:51:16 PM
UserVersion : AD Version: 0, SysVol Version: 0
WmiFilter :
...
The output has been truncated to save space, but you get the idea. We’re using the `Get-GPO` cmdlet and the `-all` parameter to list Group Policy Objects for every GPO in the `jdh1ab.local` domain. The cmdlet would have used the current, default domain, but we wanted to demonstrate how you query a specific domain. The output looks like text, but these are objects with properties, like `DisplayName` and `GPOStatus`.

---

**Leveraging Objects for Fun and Profit**

We’ll talk a lot about objects in this chapter. Let’s take a second to discuss the power of outputting an object instead of just text.

What is an object?

An object is a logical representation of a “thing.” This thing could have properties and, possibly, methods. This thing could be anything from a car to a file located on a hard drive. In PowerShell, these things are .NET classes or COM objects.

Why does this matter?

Dealing with objects instead of text gives us a huge advantage. Let’s take a car, for example. A car has a type, model, tires, color, and a slew of other properties. It also has functions (or methods), like start, stop, drive, and shift. When dealing with text, we could just have the following:

<table>
<thead>
<tr>
<th>Make</th>
<th>Type</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mazda</td>
<td>MPV</td>
<td>White</td>
</tr>
<tr>
<td>BMW</td>
<td>328i</td>
<td>Gray</td>
</tr>
<tr>
<td>Mazda</td>
<td>Mazda3</td>
<td>Gray</td>
</tr>
</tbody>
</table>

This is fine if we don’t need to do anything with this data, but what if we do? What if we just wanted to see the cars that are gray or cars that are made by Mazda? We would have to search the text and find the lines that have the information we need and manipulate the results to output what we are after.

In PowerShell, this is very simple—because each car is an object with properties that we can easily filter according to any one individual property. Let’s consider the following example.
Here’s an interesting example for us to hang our hats on. In the next section, we’ll examine the idea of objects in detail, specifically, objects returned by the Group Policy cmdlets.

Let’s say your boss comes in and asks you for a list of every GPO you have in the domain. He doesn’t want to see all the fluff, just the names. You’ll have it for him before he leaves the room by running this:

```powershell
PS C:\> Get-gpo -all | Select DisplayName
```

This line will get all the cars made by Mazda and output only the type:

```powershell
PS> Get-Cars | where{$_Make -eq 'Mazda'} | select-object
```

What if we want all the gray cars? We’d simply type this line:

```powershell
PS> Get-Cars | where-object {$_Color -eq 'Gray'}
```

As you can see, this is much easier to manage and much more powerful than messing with raw text.

All we’re doing is piping the output from Get-GPO to the select-object cmdlet (using the alias select) and stripping off everything but the DisplayName property. The boss is clearly impressed, but he says, “You know...I wonder when they were created?”

You turn around to your computer and you run this command, based on (you guessed it) the same Get-GPO cmdlet:

```powershell
PS C:\> Get-gpo -all | Select DisplayName,CreationTime
```

DisplayName
----------
Network Config
HR Desktop
Win7 Special
Remote Assistance
Sales Desktop
Default Domain Policy
New Group Policy Object
WinRM Configuration
ADMDemo
...

All we’re doing is piping the output from Get-GPO to the select-object cmdlet (using the alias select) and stripping off everything but the DisplayName property. The boss is clearly impressed, but he says, “You know...I wonder when they were created?”

You turn around to your computer and you run this command, based on (you guessed it) the same Get-GPO cmdlet:
<table>
<thead>
<tr>
<th>DisplayName</th>
<th>CreationTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Config</td>
<td>9/13/2010 3:17:41 PM</td>
</tr>
<tr>
<td>HR Desktop</td>
<td>9/2/2010 4:55:11 PM</td>
</tr>
<tr>
<td>Win7 Special</td>
<td>7/12/2010 4:05:38 PM</td>
</tr>
<tr>
<td>Remote Assistance</td>
<td>2/20/2011 7:58:59 PM</td>
</tr>
<tr>
<td>Sales Desktop</td>
<td>9/2/2010 4:54:59 PM</td>
</tr>
<tr>
<td>Default Domain Policy</td>
<td>1/24/2010 3:02:46 PM</td>
</tr>
<tr>
<td>WinRM Configuration</td>
<td>3/26/2010 8:15:42 AM</td>
</tr>
</tbody>
</table>

This time, you tell Select-Object to only select the DisplayName and CreationTime properties. Again, before the boss leaves, you get him what he asked for. As he picks his jaw off the floor, he says, “I don’t suppose you could get me all the GPOs that have been changed in the last week?” You say, “Sure, one sec.” Back to the Get-GPO cmdlet again with this command:

```powershell
PS C:\> get-gpo -all |>
>> where {$_ .ModificationTime -gt (get-date).AddDays(-7)} |>
>> Select DisplayName,ModificationTime
```

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>ModificationTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engineering Desktop</td>
<td>8/14/2012 12:07:30 PM</td>
</tr>
<tr>
<td>Corp Desktop</td>
<td>8/14/2012 12:08:52 PM</td>
</tr>
</tbody>
</table>

Your boss is clearly impressed at your speed and agility, and he doubles your salary! Okay, maybe not, but he will definitely know who has the information he needs, and he will remember that come review time.

You can type the command as one line, letting PowerShell wrap as necessary, or you can manually break the line to make it easier to read or to fit on a printed page, as we’ve done here. The best approach is to press Enter after a pipe character, an open curly brace, or an open parenthesis.
## Getting Information about Objects

In the Get-GPO examples, we used Select-Object to filter out some properties. You may be curious about how we even knew what properties were available.

The easiest way to get a list of what properties and methods an object has is to pipe the object to a cmdlet called Get-Member. Get-Member is used to “reflect” the object and list the properties and methods it finds.

Here is an example:

```powershell
PS C:\> get-gpo "Default Domain Policy" | Get-Member
```

<table>
<thead>
<tr>
<th>Name</th>
<th>MemberType</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.GpoBackup</td>
</tr>
<tr>
<td>CopyTo</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.Gpo CopyTo</td>
</tr>
<tr>
<td>Delete</td>
<td>Method</td>
<td>System.Void Delete()</td>
</tr>
<tr>
<td>Equals</td>
<td>Method</td>
<td>bool Equals(Microsoft.GroupPolicy...)</td>
</tr>
<tr>
<td>GenerateReport</td>
<td>Method</td>
<td>string GenerateReport(Microsoft...)</td>
</tr>
<tr>
<td>GenerateReportToFile</td>
<td>Method</td>
<td>System.Void GenerateReportToFile(...)</td>
</tr>
<tr>
<td>GetHashCode</td>
<td>Method</td>
<td>int GetHashCode()</td>
</tr>
<tr>
<td>GetSecurityInfo</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.GPPermission</td>
</tr>
<tr>
<td>GetType</td>
<td>Method</td>
<td>type GetType()</td>
</tr>
<tr>
<td>Import</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.Gpo Import</td>
</tr>
<tr>
<td>IsAclConsistent</td>
<td>Method</td>
<td>bool IsAclConsistent()</td>
</tr>
<tr>
<td>MakeAclConsistent</td>
<td>Method</td>
<td>System.Void MakeAclConsistent()</td>
</tr>
<tr>
<td>SetSecurityInfo</td>
<td>Method</td>
<td>System.Void SetSecurityInfo(Microsoft...)</td>
</tr>
<tr>
<td>ToString</td>
<td>Method</td>
<td>string ToString()</td>
</tr>
<tr>
<td>Computer</td>
<td>Property</td>
<td>Microsoft.GroupPolicy.ComputerCore</td>
</tr>
<tr>
<td>CreationTime</td>
<td>Property</td>
<td>System.DateTime CreationTime {get;}</td>
</tr>
<tr>
<td>Description</td>
<td>Property</td>
<td>System.String Description {get;set;}</td>
</tr>
<tr>
<td>DisplayName</td>
<td>Property</td>
<td>System.String DisplayName {get;set;}</td>
</tr>
<tr>
<td>DomainName</td>
<td>Property</td>
<td>System.String DomainName {get;}</td>
</tr>
<tr>
<td>GpoStatus</td>
<td>Property</td>
<td>Microsoft.GroupPolicy.GpoStatus G...</td>
</tr>
<tr>
<td>Id</td>
<td>Property</td>
<td>System.Guid Id {get;}</td>
</tr>
<tr>
<td>ModificationTime</td>
<td>Property</td>
<td>System.DateTime ModificationTime ...</td>
</tr>
</tbody>
</table>
We can also use `Get-GPO` to examine details about a specific GPO:

```
PS C:\> get-gpo 'default domain policy'
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisplayName</td>
<td>Default Domain Policy</td>
</tr>
<tr>
<td>DomainName</td>
<td>jdhlab.local</td>
</tr>
<tr>
<td>Owner</td>
<td>JDHLAB\Domain Admins</td>
</tr>
<tr>
<td>Id</td>
<td>31b2f340-016d-11d2-945f-00c04fb984f9</td>
</tr>
<tr>
<td>GpoStatus</td>
<td>AllSettingsEnabled</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>CreationTime</td>
<td>1/24/2010 3:02:46 PM</td>
</tr>
<tr>
<td>ModificationTime</td>
<td>6/30/2011 4:51:16 PM</td>
</tr>
<tr>
<td>UserVersion</td>
<td>AD Version: 0, SysVol Version: 0</td>
</tr>
<tr>
<td>WmiFilter</td>
<td></td>
</tr>
</tbody>
</table>

There's a lot of useful information here!

### Operators in PowerShell

In PowerShell, there are comparison operators, which are used to compare one object to another object. Understanding the operators' functionality is key. There are about 26 operators that you can use, but we want to cover only a few of them:

- `eq`: Equal
- `ne`: Not equal
- `ge`: Greater than or equal
- `gt`: Greater than
- `lt`: Less than
You'll notice that, instead of DisplayName, this cmdlet uses Name. How did we know? Simple, we used Get-Member to discover the property names for the objects that Get-GPO writes to the pipeline:

```
PS C:\> get-gpo 'default domain policy' | get-member
```

TypeName: Microsoft.GroupPolicy.Gpo

<table>
<thead>
<tr>
<th>Name</th>
<th>MemberType</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.GpoBackup ...</td>
</tr>
<tr>
<td>CopyTo</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.Gpo CopyTo(M...</td>
</tr>
<tr>
<td>Delete</td>
<td>Method</td>
<td>void Delete()</td>
</tr>
<tr>
<td>Equals</td>
<td>Method</td>
<td>bool Equals(Microsoft.GroupPolicy.Gpo...)</td>
</tr>
<tr>
<td>GenerateReport</td>
<td>Method</td>
<td>string GenerateReport(Microsoft.Gro....</td>
</tr>
<tr>
<td>GenerateReportToFile</td>
<td>Method</td>
<td>void GenerateReportToFile(Microsoft.Gr...</td>
</tr>
<tr>
<td>GetHashCode</td>
<td>Method</td>
<td>int GetHashCode()</td>
</tr>
<tr>
<td>GetSecurityInfo</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.GPPermissionCol...</td>
</tr>
<tr>
<td>GetType</td>
<td>Method</td>
<td>type GetType()</td>
</tr>
<tr>
<td>Import</td>
<td>Method</td>
<td>Microsoft.GroupPolicy.Gpo Import(Micro...</td>
</tr>
<tr>
<td>IsAclConsistent</td>
<td>Method</td>
<td>bool IsAclConsistent()</td>
</tr>
<tr>
<td>MakeAclConsistent</td>
<td>Method</td>
<td>void MakeAclConsistent()</td>
</tr>
</tbody>
</table>
Using PowerShell to Do More with Group Policy

SetSecurityInfo     Method     void SetSecurityInfo(Microsoft.GroupPolicy...
ToString            Method     string ToString()
Computer            Property   Microsoft.GroupPolicy.ComputerConfigura...
CreationTime        Property   datetime CreationTime {get;}
Description          Property   string Description {get;set;}
DisplayName          Property   string DisplayName {get;set;}
DomainName           Property   string DomainName {get;}
GpoStatus            Property   Microsoft.GroupPolicy.GpoStatus GpoStat...
Id                   Property   guid Id {get;}
ModificationTime     Property   datetime ModificationTime {get;}
Owner                Property   string Owner {get;}
Path                 Property   string Path {get;}
User                 Property   Microsoft.GroupPolicy.UserConfiguration...
WmiFilter            Property   Microsoft.GroupPolicy.WmiFilter WmiFilt...

If we look at a specific GPO, we should get essentially the same information that we saw earlier:

PS C:\> get-gpo 'Corp Desktop'

DisplayName      : Corp Desktop
DomainName       : jdhlab.local
Owner            : JDHLAB\Domain Admins
Id               : cb903cce-8e60-4e83-bbcd-944cc4ccc0eb
GpoStatus        : AllSettingsEnabled
Description      :
CreationTime     : 9/8/2010 8:58:54 AM
ModificationTime : 8/14/2012 12:08:52 PM
WmiFilter        :

We hope you realize we haven’t run a single PowerShell script. Who says using PowerShell is difficult?

Creating GPO Reports

Creating printed documentation for all your GPOs couldn’t be easier in PowerShell using the new Microsoft Get-GPOReport cmdlet. You can create a pretty HTML report or dump the data to an XML file:

PS C:\>Get-GPOReport ‘Default Domain Policy’ -ReportType HTML
-Path \File01\GPReports\DefaultDomain.html
This command will create an HTML report for the Default Domain Policy GPO, saved to the specified file. The cmdlet assumes the GPO is in the current domain, although you can specify a specific domain and/or Domain Controller.

```
PS C:\> Get-GPOReport 'Firewall Settings' -domain jdhlab.local -server DC01 -ReportType XML -Path F:\GPReports\Firewall.xml
```

The `Get-GPOReport` cmdlet includes an `–All` parameter. When used, it will create a single XML or HTML report for every GPO in your domain. If you prefer to generate reports for all GPOs, but with each GPO as a separate file, combine a few cmdlets in a single PowerShell expression:

```
PS C:\> Get-GPO -all | ForEach-Object {
    $report='F:\GPReports\{0}.htm' -f $_.displayName.Replace(" ","")
    Get-GPOReport $_.DisplayName -ReportType HTML -Path $report
}
```

Each GPO from `Get-GPO` is piped to the `ForEach-Object` cmdlet. We’re creating a variable called `$report` to indicate the file name. Using a little PowerShell sleight of hand, we’re incorporating the display name of each GPO that is piped in, stripping out any spaces. The second step in the loop calls `Get-GPOReport` and creates the report for the pipelined GPO using the variable we just created. You can view a sample HTML report in Figure BC1.4.

**Figure BC1.4**  HTML GPO report
Documenting GPO Links

In the following sections, you'll learn how to document GPO links. We’re going to find all the GPOs and their links. Again, because we're using PowerShell, we have the ability to output objects, not just straight text. Although the power of this may not immediately be evident, you'll soon see why this is such a huge benefit.

When you look through the list of new GPO cmdlets, you'll see some to create and set links like New-GPLink. But there is no cmdlet called Get-GPLink. On one hand, this makes perfect sense. Even though we can retrieve a GPO, the link is part of the Active Directory object. Fortunately, the Active Directory cmdlets return linked GPO information. Here's a PowerShell script (Listing BC1.1) you can run to get Group Policy link information for sites, domains, and organizational units. This script will require that you have configured RSAT to manage Active Directory, and you will need either a Windows Server 2008 R2 domain controller or an older DC running the ADMGS that we talked about earlier.

Listing BC1.1: Group Policy Documentation script using PowerShell

```powershell
#Requires -version 2.0
import-module grouppolicy, activedirectory
Function Get-GPOLink {

  #
  .Synopsis
  Search all organizational units and linked Group Policy Objects.
  .Description
  The Group Policy module doesn't have a Get-GPLink cmdlet. You have to search all organizational units in your domain, retrieve the LinkedGroupPolicy objects, and then find the corresponding Group Policy Objects. This function requires the ActiveDirectory and Group Policy modules to be loaded in your PowerShell session. This function only works within the current domain. It cannot search across domains.

  .Example
  PS C:\> Get-GPOLink | Sort Target
```
Retrieve a list of all GPO objects sorted by the Target property.

.example
PS C:\> Get-GPOLink | select Target,DisplayName,*Time

Search the current domain and display the linked target, and its creation and last modified times.

.inputs
None

.outputs
Custom [Microsoft.GroupPolicy.Gpo] object

.link
Get-GPO
Get-ADOrganizationalUnit
Get-ADDomain

.notes
name: Get-GPOLink
version: 2.0
author: Jeffery Hicks
lastedit: 08/14/2012

#>

[cmdletbinding()]
Param()

#nested function that does the actual work of retrieving the GPO

function Get-GPOData {

[cmdletbinding()]

Param ([string]$link,[string]$dn)

Write-Verbose "Getting $link link under $dn"
#define a REGEX pattern for a GUID
$RegEx = "\{([0-9a-fA-F]{8}([0-9a-fA-F]{4}([0-9a-fA-F]{4}([0-9a-fA-F]{12})|)\}([0-9a-fA-F]{4}([0-9a-fA-F]{4}([0-9a-fA-F]{12})|)\}([0-9a-fA-F]{4}([0-9a-fA-F]{4}([0-9a-fA-F]{12})|)\})\}";
[System.GUID]$gpoGuid=$regex.match($link).value

#parse out the domain name from the link
$dc="DC=\S*"
$a=$dc.Matches($link)[0].value
#split off the ; at the end if found. It shows up on #Site links
if ($a.IndexOf(";") -gt 0) {
    $b=$a.substring(0,$a.IndexOf(";"))
} else {
    $b=$a
}
#remove DC=
$c=$b.Replace("DC=","")

#replace comma with period to get the FQDN
$gpodomain=$c.Replace("",".")

if ($gpoGUID) {
    #get the GPO and add a custom property to show the #linked OU
    Write-Verbose 'Processing $gpoGuid'
    get-gpo -id $gpoGUID -domain $gpodomain | Add-Member -memberType NoteProperty -Name 'Target' -value $dn -passthru |
    Select-Object -property *
}
} #end Get-GPOData function

#########################################################################
# Main part of Get-GPOLink
#########################################################################

#Get Site linked GPOs
Write-Verbose 'Getting site linked GPOs'
#get configuration naming context
Write-Verbose 'Getting ConfigurationNamingContext'
$config=(get-ADRootDSE).ConfigurationNamingContext

#get sites
Write-Verbose 'Getting sites from $config'
$sites=get-adobject -LDAPFilter "(Objectclass=site)"
-searchbase $config -properties gpLink

foreach ($site in $sites) {
    Write-Verbose "Processing site $($site.name)"
    #only parse if there is a gplink value
    if ($site.gplink) {
        #split gplink since all links are passed as one string
        $links=$site.gplink.split("[")[1] | Where {$_}
        Foreach ($link in $links) {
            Write-Verbose "Getting GPO data"
            Get-GPOData $link $site.distinguishedname
        } #end foreach $link
    } #end if $site.gplink
} #end foreach $site

#Get Domain linked GPOs
$domain=Get-ADDomain
Write-Verbose 'Getting domain linked GPOs from $domain.name'

foreach ($link in $domain.LinkedGroupPolicyObjects ) {
    Write-Verbose "Getting GPO data"
    Get-GPOData $link $domain.distinguishedName
} #end foreach

#Get OU Linked GPOs
Write-Verbose 'Getting OU linked GPOs'
Get-ADOrganizationalUnit -filter * -searchBase $domain |
    foreach {
        #assign the current OU object to a variable to make
        #it easier to follow
Using PowerShell to Do More with Group Policy

$OU=$_
Write-Verbose "Processing $($OU.Distinguishedname)"
#get GPO Links for the OU
$links=$OU.LinkedGroupPolicyObjects

foreach ($link in $links) {
    Write-Verbose "Getting GPO data"
    Get-GPOData $link $ou.DistinguishedName
} #end for each link
} #end for each OU

} # End Function

If you run the script, nothing will happen. The script loads the necessary modules and then defines a function called Get-GPOLink. To use the script, you need to dot-source it in your PowerShell session or your PowerShell profile. The function includes help information just like a cmdlet. Run help get-gpolink to see for yourself.

**Dot-sourcing**

Normally, when you run a script or function, anything you define, from the function itself to variables or PSDrives, disappears when the script or function finishes. One way to keep items such as functions and variables is to dot-source the file. To accomplish this, insert a period before you run the script. Using Listing BC1.1 to load Get-GPOLink into your PowerShell session, you type this command, assuming you are in the same directory as the script file:

```
PS C:\>. .\Get-GPOLink.ps1
```

It might look a little funny, and don’t forget a space after the first period. What that dot says is, “Run this script, but keep everything it creates and defines right here.”

The function uses several Active Directory functions to discover sites, the domain, and organizational units. These Active Directory objects include a property that reflects the Group Policy link. The script does some fancy parsing of this string to pull out the GPO GUID and its domain, because it’s conceivable you have a cross-domain linked GPO! With this information, we can use Get-GPO to retrieve the object. The function returns that object with an extra property to show the connected target. By the way, this script will work with PowerShell v2 or v3.
Because the function acts like a cmdlet, you can use it in a pipelined expression like this:

```
PS C:\> Get-GPOlink | Sort Target | Select Target,DisplayName,DomainName, GPOStatus,*Time
```

<table>
<thead>
<tr>
<th>Target</th>
<th>DisplayName</th>
<th>DomainName</th>
<th>GpoStatus</th>
<th>CreationTime</th>
<th>ModificationTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>CN=Default-First-Site-Name,CN=Sites,CN=Configuration,...</td>
<td>Network Config</td>
<td>jdhlab.local</td>
<td>AllSettingsEnabled</td>
<td>9/13/2010 3:17:41 PM</td>
<td>9/13/2010 3:17:42 PM</td>
</tr>
<tr>
<td>DC=jdhlab,DC=local</td>
<td>Firewall Settings</td>
<td>jdhlab.local</td>
<td>AllSettingsEnabled</td>
<td>1/19/2011 9:17:16 AM</td>
<td>2/20/2011 10:08:25 PM</td>
</tr>
<tr>
<td>DC=jdhlab,DC=local</td>
<td>Default Domain Policy</td>
<td>jdhlab.local</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You can take these objects and do anything else you need, such as filtering further, exporting, printing, or saving to a file.

**Documenting WMI Filters and Links**

The `Get-GPO` cmdlet will return WMI filter information:

```
PS C:\> get-gpo "engineering desktop"
```

```
DisplayName      : Engineering Desktop
DomainName       : jdhlab.local
Owner            : JDHLAB\Domain Admins
Id               : 6ee6fd40-03db-4dd7-91e3-8329df33cae9
GpoStatus        : UserSettingsDisabled
Description      :
ModificationTime : 8/14/2012 12:07:30 PM
ComputerVersion  : AD Version: 2, SysVol Version: 2
WmiFilter        : Windows 7

But if you try to select just the `WmiFilter` property, you'll get a little surprise:

```
PS C:\> get-gpo "engineering desktop" | Select WmiFilter
```

```
Microsoft.GroupPolicy.WmiFilter
```

The cmdlet output is pulling the WMI filter name from the underlying object, but the property itself is another object. If you only need a single nested property like this, you can expand it:

```
PS C:\> get-gpo "engineering desktop" | Select -expandproperty WmiFilter
```
Or you can use a hash table with Select-Object to pull out the filter information nestled away:

```powershell
PS C:\> get-gpo -all | Where {$_.'WMIFilter'} |
>> Select Displayname, @(name='WMIFilter';Expression={$_..'WMIFilter'.name}),
>> @(Name='WMIDescription';Expression={$_..'WMIFilter'.Description})
```

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>WMIFilter</th>
<th>WMIDescription</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win7 Special</td>
<td>Windows 7</td>
<td>Any Win7 flavor</td>
</tr>
<tr>
<td>Engineering Desktop</td>
<td>Windows 7</td>
<td>Any Win7 flavor</td>
</tr>
<tr>
<td>XP Special</td>
<td>XP Systems</td>
<td>any flavor</td>
</tr>
</tbody>
</table>

The expression gets all GPOs and filters them using Where-Object to only keep GPOs with a WMIFilter defined. The hash tables create custom properties with values pulled from the WMIFilter object.

There's one minor “gotcha” here. This expression will return all WMI filters currently linked; it won't show you other filters you may have defined. However, you can easily get that information by querying Active Directory, assuming you have the Active Directory module loaded and the requisite R2/ADMGS domain controllers on the back end.

```powershell
PS C:\> get-adobject -LDAPFilter "(ObjectClass=msWMI-som)" -properties * |
>> Select @{$Name='GUID';Expression={$_.Name}},
>> @{$Name='Name';Expression={$_..'msWMI-Name'}},
>> @{$Name='Description';Expression={$_..'msWMI-Name'}},
>> @{$Name='WMIQuery';Expression={$_..'msWMI-Name2'.Split(';')[-2] }},
>> Created,Modified
```

<table>
<thead>
<tr>
<th>GUID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{F8531CF9-4441-4COC-A8D1-9BA7A576855A}</td>
<td>Windows 7</td>
<td>Any Win7 flavor</td>
</tr>
<tr>
<td>WMIQuery : Select * from win32_operatingsystem</td>
<td>where caption like '%Windows 7%'</td>
<td></td>
</tr>
</tbody>
</table>
Using PowerShell to Do More with Group Policy

Using the `Get-ADObject` cmdlet, you search for objects that are of the `msWMI-som` class. You need to tell the cmdlet to return all properties. You might want to do something else, but in our example we piped the results to `Select-Object` and displayed a number of properties, some of which were calculated from the raw property values. Now we can see all of our WMI filters, regardless of their link state.

**Listing GPO Permissions**

A big part of troubleshooting Group Policy is dealing with permissions on the actual Group Policy Object. Let’s see what we can do with the new `Get-GPPermissions` cmdlets:

```
PS C:\> Get-GPPermissions -name 'default domain policy' -all
```

- **Trustee**: Authenticated Users
  - **TrusteeType**: WellKnownGroup
  - **Permission**: GpoApply
  - **Inherited**: False

- **Trustee**: Domain Admins
  - **TrusteeType**: Group
  - **Permission**: GpoCustom
  - **Inherited**: False

- **Trustee**: Enterprise Admins
  - **TrusteeType**: Group
  - **Permission**: GpoCustom
  - **Inherited**: False

- **Trustee**: Help Desk
TrusteeType : Group
Permission : GpoRead
Inherited : False

Trustee : ENTERPRISE DOMAIN CONTROLLERS
TrusteeType : WellKnownGroup
Permission : GpoRead
Inherited : False

Trustee : SYSTEM
TrusteeType : WellKnownGroup
Permission : GpoEditDeleteModifySecurity
Inherited : False

Here, we can see all permissions for the Default Domain Policy. If you prefer to limit your results to a specific user or group, use the -TargetGroup and -TargetName parameters:

PS C:\> Get-GPPermission "Sales Desktop" -TargetType Group
-TargetName "Help Desk"

Here's a very simple one-liner to get permissions for all GPOs and save the results to a text file:

get-gpo -all | foreach {
    $_.Displayname
    Get-GPPermissions $_.Displayname -all
} | out-file d:\reports\GPPerms.txt

If you want to find a specific security principal, say Help Desk, you can modify this expression slightly and add a filter with Where-Object:

get-gpo -all | foreach {
    $perms=$_ | Get-GPPermissions $_.Displayname -all
    -TargetType Group -erroraction SilentlyContinue
}
if ($perms) {
    $_.DisplayName
}

This expression gets all GPOs and pipes them to ForEach-Object. Each pipelined GPO is passed to Get-GPPermissions looking for the Help Desk group. If we were looking for a domain group, we would use the domain\groupname format. Because the cmdlet will raise an exception if the group is not found, we temporarily turn off the error pipeline with the –ErrorAction parameter. If the group is found, then $perms will exist and we display the GPO’s display name.

You may also want to find GPOs where a particular user or group does not have permissions. This is a little trickier, but we won’t make you figure it out.

get-gpo -all | foreach {
    $perms=$_ | Get-GPPermissions -all
    if ($perms.trustee.name -notcontains 'Authenticated Users') {
        $_.DisplayName
    }
}

This code takes advantage of a new PowerShell v3 feature that automatically expands properties of nested objects, like the trustee object that is part of each permission. In this snippet, if Authenticated Users isn’t found PowerShell will display the name of the GPO.

### Setting GPO Permissions

The Group Policy module includes a cmdlet called Set-GPPermissions, which you can use to modify permissions for one or more GPOs in your domain. We hope it goes without saying that whenever you start messing with permissions, you should test thoroughly in a non-production environment. Also, read through all the help and examples for this cmdlet.

The cmdlet has a PermissionLevel parameter that sets the appropriate access control on the GPO. Valid values for this parameter are GpoRead, GpoApply, GpoEdit, GpoEditDeleteModifySecurity, and None.

Let’s say we have a GPO called Desktop Additions we’re developing. Here are the current permissions:

```bash
PS C:\> Get-GPPermissions "Desktop Additions" -all

Trustee     : Authenticated Users
TrusteeType : WellKnownGroup
```
Permission : GpoApply
Inherited : False

Trustee : Domain Admins
TrusteeType : Group
Permission : GpoEditDeleteModifySecurity
Inherited : False

Trustee : Enterprise Admins
TrusteeType : Group
Permission : GpoEditDeleteModifySecurity
Inherited : False

Trustee : ENTERPRISE DOMAIN CONTROLLERS
TrusteeType : WellKnownGroup
Permission : GpoRead
Inherited : False

Using Set-GPPermissions, we want to remove Authenticated Users and give the Beta Users group permissions to apply the GPO.

PS C:\> Set-GPPermissions 'Desktop Additions'
     -TargetName 'Beta Users' -TargetType Group
     -PermissionLevel GpoApply -whatif

What if: Set the permission level to GpoApply for the security group 'Beta Users' on the 'Desktop Additions' GPO in the jdhlab.local domain. (Set-GPPermissions)

Well, we didn't quite do it. We wanted to demonstrate another useful PowerShell feature, the -WhatIf parameter. Cmdlets that make changes, such as the Set and Remove cmdlets, support this parameter. Think of it as a sanity check; the cmdlet is telling you, “This is what I would have done, if you’d let me.”
Now, let's do it for real by rerunning the previous command without –WhatIf:

```powershell
PS C:\> Set-GPPermissions "Desktop Additions"
-TargetName "Beta Users" -TargetType Group -PermissionLevel GPOApply
```

Next, let's remove Authenticated Users by setting their permission level to None:

```powershell
PS C:\> Set-GPPermissions "Desktop Additions"
-TargetName "Authenticated Users" -TargetType Group
-PermissionLevel None
```

Now, when we check permissions, they are exactly what we want:

```powershell
PS C:\> get-GPPermissions "Desktop Additions" -all
```

<table>
<thead>
<tr>
<th>Trustee</th>
<th>TrusteeType</th>
<th>Permission</th>
<th>Inherited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beta Users</td>
<td>Group</td>
<td>GpoApply</td>
<td>False</td>
</tr>
<tr>
<td>Domain Admins</td>
<td>Group</td>
<td>GpoEditDeleteModifySecurity</td>
<td>False</td>
</tr>
<tr>
<td>Enterprise Admins</td>
<td>Group</td>
<td>GpoEditDeleteModifySecurity</td>
<td>False</td>
</tr>
<tr>
<td>ENTERPRISE DOMAIN CONTROLLERS</td>
<td>WellKnownGroup</td>
<td>GpoRead</td>
<td>False</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>WellKnownGroup</td>
<td>GpoEditDeleteModifySecurity</td>
<td>False</td>
</tr>
</tbody>
</table>
Manipulating GPOs with PowerShell

Another task that administrators spend a large amount of time on is the actual manipulation of GPOs themselves. But creating, linking, and backing up GPOs are all things we should be able to script. Now that we have shown you some scripts that enable you to document GPOs, you're ready to take on some of these new challenges.

Creating New GPOs

One of the most common tasks for GPO administrators is creating new GPOs. We can easily create a new GPO with the `New-GPO` cmdlet from the Group Policy module:

```
PS C:\> New-GPO "Executive Desktops" -Comment "C-level desktop configurations"
```

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>: Executive Desktops</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainName</td>
<td>: jdhlab.local</td>
</tr>
<tr>
<td>Owner</td>
<td>: JDHLAB\Domain Admins</td>
</tr>
<tr>
<td>Id</td>
<td>: 2f138247-2fe1-483a-a3dc-526dd30c0607</td>
</tr>
<tr>
<td>GpoStatus</td>
<td>: AllSettingsEnabled</td>
</tr>
<tr>
<td>Description</td>
<td>: C-level desktop configurations</td>
</tr>
<tr>
<td>CreationTime</td>
<td>: 8/14/2012 2:22:06 PM</td>
</tr>
<tr>
<td>ModificationTime</td>
<td>: 8/14/2012 2:22:06 PM</td>
</tr>
<tr>
<td>UserVersion</td>
<td>: AD Version: 0, SysVol Version: 0</td>
</tr>
<tr>
<td>ComputerVersion</td>
<td>: AD Version: 0, SysVol Version: 0</td>
</tr>
<tr>
<td>WmiFilter</td>
<td>:</td>
</tr>
</tbody>
</table>

The only part that is required is a GPO name, but we went ahead and added a comment as well. We can take this a step further and combine some of the cmdlets we’ve already looked at. Assuming the Executive Desktops GPO doesn’t exist, look at this:

```
PS C:\> New-GPO "Executive Desktops" | new-gplink -target 'ou=executive,ou=desktops,dc=jdhlab,dc=local' | set-gppermissions -permissionlevel gpoedit -targetname 'JDHLAB\IT Admins' -targettype group
```

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>: Executive Desktops</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainName</td>
<td>: jdhlab.local</td>
</tr>
<tr>
<td>Owner</td>
<td>: JDHLAB\Domain Admins</td>
</tr>
<tr>
<td>Id</td>
<td>: f6266c2d-d5d2-4663-81cb-538835e2e3c9</td>
</tr>
<tr>
<td>GpoStatus</td>
<td>: AllSettingsEnabled</td>
</tr>
<tr>
<td>Description</td>
<td>: C-level desktop configurations</td>
</tr>
</tbody>
</table>
With one command, we created a new GPO, linked it to an OU, and added the IT Admins group as GPO editors. That’s the power of the PowerShell pipeline!

Modifying GPO Settings

Microsoft doesn’t offer much in the way of cmdlets to define settings within a GPO. Even with all this PowerShell scripting power, you still (mostly) need the GPMC for that.

However, there are two cmdlets you can use in the new GroupPolicy module to set and get Registry settings. In the GPMC, you can define Registry values for a GPO’s computer and user nodes. You can make the Registry value a policy setting or a preference. Which you use depends on your needs, but you can manage either with PowerShell.

Get-GPPrefRegistryValue

The Get-GPPrefRegistryValue cmdlet will return Registry information in a GPO, configured as a preference. You need to specify the GPO name, the GPO context, and the Registry key path:

```
PS C:\> Get-GPprefRegistryValue -Name 'corp Desktop' -Key 'HKCU\CorpData' -Context 'User'
```

<table>
<thead>
<tr>
<th>KeyPath</th>
<th>CorpData</th>
</tr>
</thead>
<tbody>
<tr>
<td>FullKeyPath</td>
<td>HKEY_CURRENT_USER\CorpData</td>
</tr>
<tr>
<td>Hive</td>
<td>CurrentUser</td>
</tr>
<tr>
<td>Action</td>
<td>Create</td>
</tr>
<tr>
<td>Order</td>
<td>1</td>
</tr>
<tr>
<td>DisabledDirectly</td>
<td>False</td>
</tr>
<tr>
<td>DisabledByAncestor</td>
<td>False</td>
</tr>
<tr>
<td>Value</td>
<td>GB1878</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
<tr>
<td>ValueName</td>
<td>Cost Center</td>
</tr>
<tr>
<td>HasValue</td>
<td>True</td>
</tr>
</tbody>
</table>

```
PS C:\> Get-GPprefRegistryValue -Name 'corp Desktop' -Key 'HKCU\CorpData' -Context 'User'
```

<table>
<thead>
<tr>
<th>KeyPath</th>
<th>CorpData</th>
</tr>
</thead>
<tbody>
<tr>
<td>FullKeyPath</td>
<td>HKEY_CURRENT_USER\CorpData</td>
</tr>
<tr>
<td>Hive</td>
<td>CurrentUser</td>
</tr>
<tr>
<td>Action</td>
<td>Update</td>
</tr>
</tbody>
</table>

```
As you can see, we have retrieved the Registry GPPreferences items in the “Alpha Test GPO” that are set under HKCU\MyCompany. Figure BC1.5 shows the same settings in the GPMC.

**FIGURE BC1.5** Group Policy Registry preferences

If you know the exact value name, you can ask for it alone:

```
PS C:\> (Get-GPPrefRegistryValue 'Corp Desktop' -Context "User" -Key "HKCU\CorpData" -ValueName "Division").value
Ops
```

**Set-GPPrefRegistryValue**

More than likely, you might want to modify or create a Registry preference. You can accomplish this task with `Set-GPPrefRegistryValue`. As with its counterpart, `Get-GPPrefRegistryValue`, you will need to supply a GPO name, a context, and the Registry path. You will also need to define a ValueName, the assigned value, its type, and an action (Create, Replace, Update, or Delete):

```
Set-GPPrefRegistryValue 'Corp Desktop' -Context User -Key 'HKCU\CorpData' -ValueName EVP -Value "Jeremy Moskowitz" -Type String -Action Create
```
Using PowerShell to Do More with Group Policy

Normally, you will want to use String or Dword, but other possible choices are ExpandString, Binary, MultiString, and Qword. In this example, we’ve added another Registry preference under HKCU\CorpData called EVP. Let’s verify:

PS C:\> Get-GPPrefRegistryValue 'Corp Desktop'
-Context 'User' -ValueName 'EVP' -Key 'HKCU\CorpData'

KeyPath : CorpData
FullPath    : HKEY_CURRENT_USER\CorpData
Hive        : CurrentUser
Action      : Create
Order       : 3
DisabledDirectly : False
DisabledByAncestor : False
Value       : Jeremy Moskowitz
Type        : String
ValueName   : EVP
HasValue    : True

You can manage any GPO Registry preference with these PowerShell cmdlets from the GroupPolicy module or the GPMC.

Using PowerShell is exactly like using the “click-click-click” method of the Group Policy Preferences Registry extension. Anything you can do with “click-click-click,” you can do with Set-GPPrefRegistryValue.

Neat!

Get-GPRegistryValue
You can also define Registry values as part of a GPO policy. To retrieve existing settings, you need to know at least the start of the Registry path. Let’s find the preference for the screen save timeout that is enabled in the Corp Desktop GPO:

PS C:\> Get-GPRegistryValue 'Corp Desktop'
-Key 'HKCU\Software\Policies\Microsoft\Windows''

KeyPath         : Software\Policies\Microsoft\Windows\Control Panel
FullPath        : HKEY_CURRENT_USER\Software\Policies\Microsoft\Windows\Cont...
Hive            : CurrentUser

KeyPath         : Software\Policies\Microsoft\Windows\PowerShell
FullPath        : HKEY_CURRENT_USER\Software\Policies\Microsoft\Windows\Powe...
Hive            : CurrentUser
Within this particular key, two settings are identified. We'll choose the one with Desktop and revise our command:

```
PS C:\> Get-GPRegistryValue "Corp Desktop"
-Key "HKCU\Software\Policies\Microsoft\Windows\Control Panel\Desktop"
```

KeyPath     : Software\Policies\Microsoft\Windows\Control Panel\Desktop
FullKeyPath : HKEY_CURRENT_USER\Software\Policies\Microsoft\Windows\Cont...
Hive        : CurrentUser
PolicyState : Set
Value       : 600
Type        : String
ValueName   : ScreenSaveTimeOut
HasValue    : True

You can see this is the same value when looking at the GPO in the GPMC, as shown in Figure BC1.6.

**FIGURE BC1.6** GPO Registry policy
**Set-GPRegistryValue**

Setting Registry values as a policy can also be done with PowerShell, but you need to think ahead. If you know the Registry key that is being modified when you configure something in a GPO, you can use `Set-GPRegistryValue` and configure that setting. For example, since we know the Registry key and values for the screen saver timeout, we can modify them with PowerShell. The syntax is similar to `Set-GPPrefRegistryValue`:

```
Set-GPRegistryValue "Corp Desktop"
-key 'HKCU\Software\Policies\Microsoft\windows\control panel\desktop'
-ValueName ScreenSaveTimeOut -Value 900 -type string
```

You can see the changed result in the GPO report in Figure BC1.7.

**FIGURE BC1.7** GPO Registry policy change

![GPO Registry policy change](image_url)

But wait, there's more. You can also set a Registry policy that doesn’t exist in the Group Policy editor. Here's how that might happen.

Suppose you have a home-grown custom application and you want to set some Registry values for that application using Group Policy.

It's easy—here's one way:

```
Set-GPRegistryValue "Corp Desktop"
-key 'HKLM\Software\MyCompany\CustomApp'
```
-ValueName InstallPath -Value '\\file02\installs\customapp.msi'
-Type String

We can verify this with Get-GPRegistryValue:

PS C:\> Get-GPRegistryValue "Corp Desktop"
-key "HKLM\Software\MyCompany\CustomApp"

KeyPath : Software\MyCompany\CustomApp
FullKeyPath : HKEY_LOCAL_MACHINE\Software\MyCompany\CustomApp
Hive : LocalMachine
PolicyState : Set
Value : '\\file02\installs\customapp.msi'
Type : String
ValueName : InstallPath
HasValue : True

Great. We can see the data is set and ready to be deployed via Group Policy. But what's happening under the hood? How does the GPMC report this? Look at Figure BC1.8.

**FIGURE BC1.8** GPO custom Registry policy setting
The setting shows up in the report as “Extra Registry Settings” because the GPMC doesn’t have any ADM or ADMX file to “match” the setting with.

To be clear: the setting will be deployed just fine. But the only way you can actually edit the setting again is via PowerShell. Alternatively, you could create a custom ADM or ADMX template for your application settings to match it up. But then, what would be the point?

This can get a little confusing, and Microsoft’s terminology doesn’t make it any easier.

---

### Policy vs. Preferences Using PowerShell Cheat Sheet

If your head is spinning about which to use—`Set-GPRegistryValue` or `Set-GPPrefRegistryValue`—here’s your cheat sheet.

- In both cases: Registry data can be created and modified using PowerShell.
- Using GPPrefs and `Set-GPPrefRegistryValue`, data can be modified directly using the Group Policy editor and viewed correctly in the GPMC GPO reports.
- Using Group Policy “Registry” settings, data cannot be modified directly using the Group Policy editor (unless there’s a matching ADM or ADMX template). Likewise, reporting shows only “Extra Registry Settings” unless there’s a matching ADM or ADMX template.

---

### Linking a GPO

We created a GPO, but what good is a GPO if you can’t link it? We briefly showed you this earlier, but let’s formally meet `New-GPLink` and `Set-GPLink`. The former will link a GPO with a target:

```powershell
PS C:\> new-gplink 'Scratch GPO'
-Target 'OU=Development,DC=mycompany,DC=local'
```

This cmdlet also has parameters that take Yes/No values: `-Enforced` and `-LinkEnabled`. They should be self-explanatory.

If you already have a link and want to modify it, you can use `Set-GPLink`. Let’s enforce the link we just created:

```powershell
PS C:\> set-gplink "Scratch GPO"
-Target "OU=Development,DC=mycompany,DC=local"
-enforced yes
```

The syntax is very similar. Remember, if you forget how to use a cmdlet, ask PowerShell for help.

```powershell
PS C:\> help set-gplink -full
```
Removing a GPO Link

We bet by now you can make a pretty good guess about what cmdlet to use to remove a GPO link:

```powershell
PS C:\> remove-gplink "Scratch GPO"
-Target "OU=Development,DC=mycompany,DC=local"
```

This cmdlet also supports `–Whatif` so you can avoid shooting yourself in the foot.

Backing Up Group Policy Objects

Another really important, but often overlooked, task is backing up GPOs. Don’t be caught without a backup of your GPOs when a disaster hits. In fact, wouldn’t it be nice to have an automated way to back up all your GPOs every night?

Backing Up One or All GPOs

The `Backup-GPO` cmdlet does exactly what its name implies. All you need to do is specify a GPO and a backup path. But here’s the gotcha: the path is relative to wherever you are running the cmdlet. If you are managing Group Policy from your Windows 8 desktop, you’ll need to reference a UNC path like this:

```powershell
PS C:\> backup-gpo "Desktop Additions"
-path '\mycompany-dc01\f$\gpobackup'
-command "Baseline backup"
```

The cmdlet makes it ridiculously easy to back up all your GPOs in one fell swoop:

```powershell
PS C:\> backup-gpo -All -path '\\mycompany-dc01\f$\gpobackup'
-comment "Weekly Backup"
```

The only caveat is that the target folder must already exist.

Restoring a GPO

What good are backups if you can’t restore them? This is easily accomplished with `Restore-GPO`. Specify the name of the GPO and the backup location:

```powershell
PS C:\> restore-gpo "Scratch GPO"
-Domain mycompany.local
-path '\\mycompany-dc01\f$\gpobackup'
```

When you specify a GPO by name, the `Restore-GPO` cmdlet will restore the most recent version. If you want any other version, you’ll need to specify it by its GUID. One major caveat: if the GPO was deleted from your domain, you can’t use this cmdlet. We’re not sure if this is a bug or a “feature,” but if the GPO doesn’t exist and you attempt to restore from backup, you’ll get a missing file error. Fortunately you can restore the GPO using the Microsoft Management Console (MMC), so we can only hope that this cmdlet will be “improved” at some future date.
Importing GPOs

What if you have a GPO from someone else’s domain or a test lab domain and want to get it into your domain? You can back it up and then import it. We even have the ability to use migration tables in this situation. We discussed migration tables at length in Chapter 2 in the section “Copy and Import with Migration Tables.” But in short, a migration table is a mapping for UNC, SIDs, and other domain-specific information that may be different in the destination domain.

PS C:\> import-gpo -BackupGpoName 'Test GPO' -TargetName 'Test GPO' -path "\\mycompany-dc01\f$\gpos.backup"

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>Test GPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainName</td>
<td>mycompany.local</td>
</tr>
<tr>
<td>Owner</td>
<td>MYCOMPANY\Enterprise Admins</td>
</tr>
<tr>
<td>Id</td>
<td>751dc9a3-1a38-4a9e-8e99-37ead76813d2</td>
</tr>
<tr>
<td>GpoStatus</td>
<td>AllSettingsEnabled</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>CreationTime</td>
<td>10/25/2009 9:56:52 PM</td>
</tr>
<tr>
<td>ModificationTime</td>
<td>10/25/2009 10:10:19 PM</td>
</tr>
<tr>
<td>UserVersion</td>
<td>AD Version: 1, SysVol Version: 1</td>
</tr>
<tr>
<td>ComputerVersion</td>
<td>AD Version: 3, SysVol Version: 3</td>
</tr>
</tbody>
</table>

In this example, we are importing a GPO from a backup set that was previously deleted. We’re using the backed-up GPO name as the new or target name. We could just as easily import a backed-up GPO from our production environment into our test lab.

If you need to use a migration table, all you need to do is specify it:

PS C:\> Import-GPO -BackupGpoName ForeignGPO -TargetName NewTestGPO -Path X:\GPO\Backups -MigrationTable X:\GPO\Migration1.migtable

Performing a Remote GPUpdate (Invoking GPUpdate)

Up until this point all the Group Policy cmdlets were available since Windows 7.

If you’ll recall in Chapter 3 in the “Manually Forcing Background Policy Processing (Remote GPUpdate)” section, you learned that the Windows 8 GPMC has a function where you can right-click on an OU and specify which computers will receive GPUpdate.

Well, there’s a PowerShell cmdlet for that too: Invoke-GPUpdate. In order for it to be used, the same rules as running the command from the GPMC apply. So, let’s review:

- The target machine must be Windows 7 or Windows 8.
- The machine must be on.
- The firewall has to have ports 135 and 445 opened up.
For more information on that last one, see the sidebar “Under the Hood with Remote Group Policy Processing” in Chapter 3.

Now, let’s take a look at the `Invoke-GPUpdate` cmdlet to see how it works.

First, you can use this cmdlet in place of the traditional `GPUpdate.exe` and force an update of the local machine:

```
PS C:\> invoke-gpupdate
```

If you prefer to limit the refresh to either user or computer settings, then specify it:

```
PS C:\> invoke-gpupdate -Computer client8 -Target User
```

This will remotely update GPO user settings on Client8. You can also apply an update to multiple computers using any of these techniques:

```
PS C:\> "client1","client2","client3" | invoke-gpupdate -Target Computer
PS C:\> get-content c:\work\computers.txt | invoke-gpupdate -Computer -Target computer
```

```
PS C:\> get-adcomputer -filter " -SearchBase
'OU=Desktops,DC=jdhlab,DC=local" |
foreach { Invoke-GPUpdate -comp $_.name -randomdelay 5}
```

The last example will get all computers in the Desktops OU and update Group Policy settings using a delay of 5 minutes plus a random offset.

This cmdlet also has parameters to force a computer to reboot (-Boot) or log off the user (-Logoff), but use these with caution. Be sure to take the time to read full help and examples for this cmdlet and test in a nonproduction setting.

**Replacing Microsoft’s GPMC Scripts with PowerShell Equivalents**

Microsoft has an army of interesting Visual Basic scripts that perform many of the functions of the GPMC. You can download the original Microsoft GPMC scripts at http://tinyurl.com/23xfz3.

However, you’ll be better served by using the new PowerShell equivalents. Table BC.1 shows the original Microsoft script, what it does, and the equivalent PowerShell cmdlets. Some of the cmdlets may need to be part of an expression using standard PowerShell cmdlets such as Where-Object and Select-Object.
# Table B.C.1 PowerShell equivalents for Microsoft scripts

<table>
<thead>
<tr>
<th>Microsoft script</th>
<th>What it does</th>
<th>PowerShell cmdlet</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackupAllGPOs.wsf</td>
<td>Backs up all GPOs</td>
<td>Backup-GPO</td>
</tr>
<tr>
<td>BackupGPO.wsf</td>
<td>Backs up a GPO</td>
<td>Backup-GPO</td>
</tr>
<tr>
<td>CopyGPO.wsf</td>
<td>Copies a GPO</td>
<td>Copy-GPO</td>
</tr>
<tr>
<td>CreateGPO.wsf</td>
<td>Creates a GPO</td>
<td>New-GPO</td>
</tr>
<tr>
<td>DeleteGPO.wsf</td>
<td>Deletes a GPO</td>
<td>Remove-GPO</td>
</tr>
<tr>
<td>DumpGPOInfo.wsf</td>
<td>Dumps GPO information</td>
<td>Get-GPO</td>
</tr>
<tr>
<td>FindDisabledGPOs.wsf</td>
<td>Finds all disabled GPOs</td>
<td>Get-GPO</td>
</tr>
<tr>
<td>FindDuplicateNamedGPOs.wsf</td>
<td>Finds duplicate named GPOs</td>
<td>Get-GPO</td>
</tr>
<tr>
<td>FindGPOsByPolicyExtension.wsf</td>
<td>Finds GPOs by policy extension</td>
<td>None</td>
</tr>
<tr>
<td>FindGPOsBySecurityGroup.wsf</td>
<td>Finds GPOs by security group</td>
<td>Get-GPPermissions</td>
</tr>
<tr>
<td>FindGPOsWithNoSecurityFiltering.wsf</td>
<td>Finds GPOs with no security filtering</td>
<td>Get-GPPermissions</td>
</tr>
<tr>
<td>findorphanedGPOsInSYSVOL.wsf</td>
<td>Finds orphaned GPOs in SYSVOL</td>
<td>None</td>
</tr>
<tr>
<td>FindUnlinkedGPOs.wsf</td>
<td>Finds GPOs with no links</td>
<td>None</td>
</tr>
<tr>
<td>GetReportsForAllGPOs.wsf</td>
<td>Gets reports for all GPOs</td>
<td>Get-GPOReport</td>
</tr>
<tr>
<td>GetReportsForGPO.wsf</td>
<td>Gets reports for a GPO</td>
<td>Get-GPOReport</td>
</tr>
<tr>
<td>GrantPermissionOnAllGPOs.wsf</td>
<td>Grants the permissions for all GPOs</td>
<td>Set-GPPermissions</td>
</tr>
<tr>
<td>ImportAllGPOs.wsf</td>
<td>Imports all GPOs in a path</td>
<td>Import-GPO</td>
</tr>
<tr>
<td>ImportGPO.wsf</td>
<td>Imports a GPO in a path</td>
<td>Import-GPO</td>
</tr>
</tbody>
</table>
TABLE BC.1  PowerShell equivalents for Microsoft scripts (continued)

<table>
<thead>
<tr>
<th>Microsoft script</th>
<th>What it does</th>
<th>PowerShell cmdlet</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListAllGPOs.wsf</td>
<td>Lists all GPOs</td>
<td>Get-GPO</td>
</tr>
<tr>
<td>QueryBackupLocation.wsf</td>
<td>Lists all GPOs in a backup</td>
<td></td>
</tr>
<tr>
<td>RestoreAllGPOs.wsf</td>
<td>Restores all GPOs in a path</td>
<td>Restore-GPO</td>
</tr>
<tr>
<td>RestoreGPO.wsf</td>
<td>Restore a GPO in a path</td>
<td>Restore-GPO</td>
</tr>
</tbody>
</table>

Any scripts you need can be found in the Book Resources section of www.GPanswers.com, as well as any updates and notes if they’re available.

Final Thoughts

Here’s a great idea! Let’s manually configure the permissions on 500 GPOs and adjust them to our needs. Or document them. Or back them up. If that sounds like a good time to you, then knock yourself out.

But we think you’re smarter than that and realize how much time and energy you can save with PowerShell. In fact, if you look around we’re sure you’ll find some cool stuff from vendors like SpecOps Software and SDM Software that offer PowerShell-centered solutions. Granted, their offerings may require a financial investment, but when you think about the investment you’ve already made in learning PowerShell it is a pretty good deal.

You can see a talk Jeff and I did at TechEd 2012 called “Group Policy Reporting and Analysis with Windows PowerShell” here:


If you are curious about learning more about Windows PowerShell, pick up a copy of *PowerShell in Depth* by Don Jones, Richard Siddaway, and Jeffery Hicks (Manning Publications, 2012). Or grab a copy of Jeff’s book *Managing Active Directory with Windows PowerShell: TFM 2nd Ed.*, (Sapien Technologies, 2011).