Microsoft

Windows PowerShell 3.0

First Steps

Ed Wilson

Windows PowerShell 3.0 First Steps

Get started with this powerful Windows administration tool

Automate Windows administration tasks with ease by learning the fundamentals of Windows PowerShell 3.0. Led by a Windows PowerShell expert, you'll learn must-know concepts and techniques through easy-to-follow explanations, examples, and exercises. Once you complete this practical introduction, you can go deeper into the Windows PowerShell command line interface and scripting language with *Windows PowerShell 3.0 Step by Step*.

Discover how to:

- Create effective Windows PowerShell commands with one line of code
- Apply Windows PowerShell commands across several Windows platforms
- Identify missing hotfixes and service packs with a single command
- Sort, group, and filter data using the Windows PowerShell pipeline
- Create users, groups, and organizational units in Active Directory
- Add computers to a domain or workgroup with a single line of code
- Run Windows PowerShell commands on multiple remote computers
- Unleash the power of scripting with Windows Management Instrumentation (WMI)

About the Author

Ed Wilson, a senior consultant at Microsoft Corporation, is a scripting expert who delivers workshops to Microsoft customers and employees worldwide. His books on Windows scripting include *Windows PowerShell 2.0 Best Practices* and *Microsoft VBScript Step by Step*. Ed also writes the popular TechNet blog, "Hey, Scripting Guy!"

ISBN: 978-0-7356-8100-2

9 780735 681002 90000

U.S.A. $29.99
Canada $31.99
[Recommended]

Operating Systems/Windows Server

Microsoft

Microsoft

Windows PowerShell 3.0 First Steps

Ed Wilson

ISBN: 978-0-7356-8100-2

1 2 3 4 5 6 7 8 9 LSI 8 7 6 5 4 3

Printed and bound in the United States of America.

Acquisitions and Developmental Editor: Michael Bolinger

Production Editor: Melanie Yarbrough

Editorial Production: Box Twelve Communications

Technical Reviewer: Brian Wilhite

Indexer: Box Twelve Communications

Cover Design: Twist Creative • Seattle

Cover Composition: Ellie Volckhausen

Illustrator: Rebecca Demarest

To Teresa, my soul mate.

—Ed Wilson

Contents at a glance

Contents

Foreword

There are many reasons to get started with automation. For me it was a little turtle from a program called LOGO. Of course, at the time I had no idea I was learning programming. I was just a kid in elementary school having fun, drawing little pictures. Years later, I became an IT administrator and developed an aversion to tedious tasks, such as manually copying a file to 100 remote servers. I started automating because I just couldn't stand the thought of repeating monotonous tasks over and over again. It took a while before I connected the dots and realized that the little turtle had paved the way for a career focused on using and teaching automation.

Windows PowerShell has really hit a sweet spot with automation in the Windows universe, balancing powerful and far-reaching capabilities while remaining simple enough that someone without deep technical expertise can start taking advantage of it quickly. Though Windows PowerShell can be a simple automation environment, it has nuances that can make it a bit tricky to really master, akin to driving a car with a manual transmission. It might be tricky to get started, but once the car is moving in first gear, the rest comes pretty easily. Ed Wilson has done a wonderful job in this book getting you started in Windows PowerShell, providing simple, prescriptive guidance to get you into first gear quickly.

As a Senior Premier Field Engineer and a Windows PowerShell Technology Lead for Microsoft Services, I spend most of my days in front of Microsoft's customers trying to teach them Windows PowerShell and hopefully getting them to love Windows PowerShell as much as I do. In every class I teach, I can't stress enough the return on investment (ROI) you get from learning Windows PowerShell. It never ceases to amaze me how once you grasp the core concepts of Windows PowerShell, you can apply them over and over again to get so much business value and personal satisfaction.

One point I try to make during every class I teach is that the words "Windows PowerShell" and "scripting" can most definitely be mutually exclusive. Technically speaking, Windows PowerShell one-liners are still "scripts," but to me they strike a nice balance between the creation of solutions and the need for developer-oriented skills. One-liners are usually very task-oriented and logically simple, yet they can accomplish a staggering amount of automation. Those who are just getting started with Windows PowerShell will find that they can become great at Windows PowerShell without writing scripts. Throughout much of this book, Ed has focused on the concepts and simplicity of Windows PowerShell. He doesn't talk directly about scripting until late in the book. Ultimately, scripting and tool-making become parts of the advanced user's skill set, but you can go a long way before that needs to happen.

No matter how diverse the skill set of my students, there is something for everyone in my classroom. Windows PowerShell has been created in such a way that it can be fun and effective for everyone from the IT novice to the expert developer. For example, the fact that it is fully object-based and sits on top of the .NET Framework is a detail that pure beginners might have no knowledge of. They can go about their Windows PowerShell days simply running commands, never really digging into the object model, but still implement valuable automation. The day they learn about objects, they can start to unlock so much more. The fact that Windows PowerShell can appeal to such diverse skills levels simultaneously is amazing to me.

When I really think about the value of Windows PowerShell and why someone new to it should dive right in, I think about the fundamental comparison of "creation" vs. "operation." By over-simplifying the roles in IT, you can see a dividing line between developers and administrators. Developers are creating solutions, and administrators are managing the design, deployment, and operation of the systems used in the process. Windows PowerShell can bridge the dividing gap to link it all together. It also allows administrators to create automation solutions without needing a true developer. There are enough elements in the Windows PowerShell language that hide and simplify the true complexity that lurks under the surface, allowing IT pros to be more effective and valuable in the workplace. Learning Windows PowerShell is an incredibly powerful tool that will truly make you more valuable to your business and often make your life easier in the process.

Ed "The Scripting Guy" Wilson is what some people call a "PowerShellebrity." He's a superstar in the Windows PowerShell world, has extensive scripting experience, and is one of the most energetic and passionate people I have ever met. I am grateful that Ed writes these books because it allows so many people access to his extensive experience and knowledge. This book is such a concise and easy way to get started with Windows PowerShell, I can't imagine putting it down if I were a beginner. Whether you have already started your Windows PowerShell journey or are just getting started, this book will help define your next steps with Windows PowerShell.

—Gary Siepsert
Senior Premier Field Engineer (PFE)
Microsoft Corporation

Introduction

Gary said nearly everything I wanted to include in the Introduction. I designed this book for the complete beginner, and you should therefore read the book from beginning to end. If you want a more reference oriented book, you should check out my PowerShell Best Practices books, or even *PowerShell 3.0 Step by Step*. Actually, the Step by Step book is not really a reference, but a hands-on learning guide. It is, ideally, the book you graduate to once you have completed this one. For your daily dose of PowerShell, you should check out my Hey Scripting Guy blog at *www.ScriptingGuys.com/blog*. I post new content there twice a day.

System Requirements

Hardware Requirements

Your computer should meet the following minimum hardware requirements:

- 2.0 GB of RAM (more is recommended)
- 80 GB of available hard disk space
- Internet connectivity

Software Requirements

To complete the exercises in this book, you should have Windows PowerShell 3.0 installed:

- You can obtain Windows PowerShell 3.0 from the Microsoft Download Center by downloading the Windows Management Framework and installing it on either Windows 7 Service Pack 1, Windows Server 2008 R2 SP1, or Windows Server 2008 Service Pack 2.
- Windows PowerShell 3.0 is already installed on Windows 8 and on Windows Server 2012. You can obtain evaulation versions of those operating systems from TechNet:

  http://technet.microsoft.com/en-US/evalcenter/hh699156.aspx?ocid=wc-tn-wctc

  http://technet.microsoft.com/en-US/evalcenter/hh670538.aspx?wt.mc_id=TEC_108_1_4
- The section on Active Directory requires access to Active Directory Domain Services. For those examples, ensure you have access to Windows Server 2012.
- For the chapter on Exchange server, you need access to a server running Microsoft Exchange Server 2013. You can obtain an evaluation version of that from TechNet:

  http://technet.microsoft.com/en-us/evalcenter/hh973395.aspx

Acknowledgments

Many people contributed the success of this book. The first person is Teresa Wilson, aka "The Scripting Wife." She is always my first reader, and nothing leaves the house without her approval. Second, I must mention my tech reviewer, Brian Wilhite, who did a great job of catching bugs, errors, and things that are misleading. I also want to thank the Charlotte PowerShell User Group whose questions, comments, and the like contributed in a significant way to the book. I kept them in mind as I wrote. I also want to thank Michael Bolinger and Melanie Yarbrough from O'Reilly for doing a great job seeing this project to completion.

Support & Feedback

The following sections provide information on errata, book support, feedback, and contact information.

Errata

We have made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://aka.ms/WinPS3FS/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, please email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let us keep the conversation going! We are on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 1

Overview of Windows PowerShell 3.0

- Understanding Windows PowerShell
- Working with Windows PowerShell
- Using Windows PowerShell cmdlets
- Supplying options for cmdlets
- Working with Help options

When you first start Windows PowerShell, whether it is the Windows PowerShell console or the Windows PowerShell Integrated Scripting Environment (ISE), the blank screen simply waits for your command. The problem is there are no hints as to what that command might be. There are no wizards or other Windows types of features to guide you in using the shell.

The name is Windows PowerShell for two reasons: It is a shell, and it is powerful. It is a mistake to think that Windows PowerShell is simply a scripting language because it is much more than that. In the same way, it is a mistake to think that Windows PowerShell is limited to running only a few cmdlets. Through scripting, it gains access to the entire realm of management technology available in the Windows world.

This chapter introduces you to Windows PowerShell and illustrates the incredible power available to you from this flexible and useful management tool.

Understanding Windows PowerShell

Windows PowerShell comes in two flavors. The first is an interactive console (similar to a KORN or BASH console in the UNIX world) built into the Windows command prompt. The Windows PowerShell console makes it simple to type short commands and to receive sorted, filtered, and formatted results. These results easily display to the console but also can redirect to .xml, .csv, or text files. The Windows PowerShell console offers several advantages such as speed, low memory overhead, and a comprehensive transcription service that records all commands and command output.

The other flavor of Windows PowerShell is the Windows PowerShell ISE. The Windows PowerShell ISE is an Integrated Scripting Environment, but this does not mean you must use it to write scripts. In fact, many Windows PowerShell users like to write their code in the Windows PowerShell ISE to take advantage of syntax coloring, drop-down lists, and automatic parameter revelation features.

In addition, the Windows PowerShell ISE has a feature called Show Command Add-On that allows you to use a mouse to create Windows PowerShell commands from a graphical environment. Once you create the command, the command either runs directly or is added to the Script pane. The choice is up to you. For more information about using the Windows PowerShell ISE, see Chapter 10, "Using the Windows PowerShell ISE."

NOTE **When I work with single commands, for simplicity I show the command and results from within the Windows PowerShell console. But keep in mind that all commands also run from within the Windows PowerShell ISE. Whether the command runs in the Windows PowerShell console, in the Windows PowerShell ISE, as a scheduled task, or as a filter for Group Policy, Windows PowerShell is Windows PowerShell is Windows PowerShell. In its most basic form, a Windows PowerShell script is simply a collection of Windows PowerShell commands.**

Working with Windows PowerShell

Windows PowerShell 3.0 is included on Windows 8 and Windows Server 2012. On Windows 8, you need only type the first few letters of the word *PowerShell* in the Start window before Windows PowerShell appears as an option. Figure 1-1 illustrates this point. I typed only **pow** in the Search box before the Start window offered Windows PowerShell as an option.

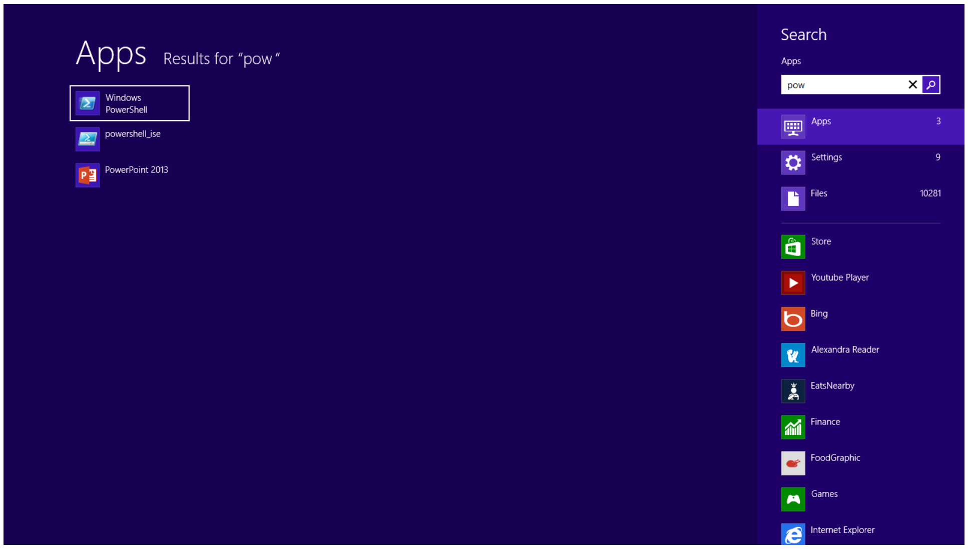


FIGURE 1-1 Typing in the Start window opens the Search window highlighting the Windows PowerShell console.

Because navigating to the Start window and typing **pow** each time I want to launch Windows PowerShell is a bit cumbersome, I prefer to pin shortcuts to the Windows PowerShell console (and the Windows PowerShell ISE) to both the Start window and the Windows taskbar. This technique of pinning shortcuts to the applications, as shown in Figure 1-2, provides single-click access to Windows PowerShell from wherever I might be working.

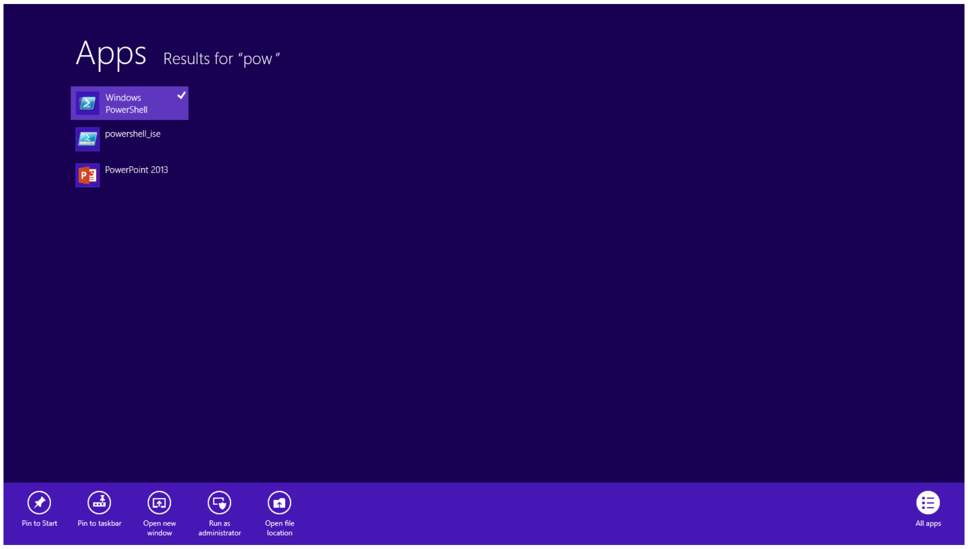


FIGURE 1-2 By right-clicking the Windows PowerShell icon in the Search results box, the Pin to Start and the Pin to taskbar options appear.

On Windows Server 2012, it is unnecessary to find the icon by using the Search box on the Start window because an icon for the Windows PowerShell console exists by default on the taskbar of the desktop.

NOTE **The Windows PowerShell ISE (the script editor) does not exist by default on Windows Server 2012. You need to add the Windows PowerShell ISE as a feature. I show how to use the Windows PowerShell ISE in Chapter 10, "Using the Windows Powershell ISE."**

Security issues with Windows PowerShell

There are two ways to launch Windows PowerShell: as an administrator or as a normal, or non-elevated, user. As a best practice, start Windows PowerShell with minimum rights. On Windows 7 and Windows 8, this means simply clicking on the Windows PowerShell icon. It opens as a non-elevated user, even if you are logged on with administrator rights. On Windows Server 2012, Windows PowerShell automatically launches with the rights of the current user. Therefore, if you are logged on as a domain administrator, the Windows PowerShell console launches with domain administrator rights.

Running as a non-elevated user

Because Windows PowerShell adheres to Windows security constraints, a user of Windows PowerShell cannot do anything the user account does not have permission to do. Therefore, if you are a non-elevated user, you do not have rights to perform tasks such as installing printer drivers, reading from the Security Log, or changing the system time.

If you are an administrator on a local Windows 7 or Windows 8 computer and you do not launch Windows PowerShell with administrator rights, you will get errors when you attempt to take certain actions, such as viewing the configuration of your disk drives. The following example shows the command and associated error:

```
PS C:\> get-disk
get-disk : Access to a CIM resource was not available to the client.
At line:1 char:1
+ get-disk
+ ~~~~~~~~
    + CategoryInfo          : PermissionDenied: (MSFT_Disk:ROOT/Microsoft/Windows/S
   torage/MSFT_Disk) [Get-Disk], CimException
    + FullyQualifiedErrorId : MI RESULT 2,Get-Disk
```

TIP **If you attempt to run cmdlets that require elevated rights, you will encounter inconsistencies with errors. For example, in a non-elevated Windows PowerShell console, the error from the *Get-Disk* cmdlet is Access To A CIM Resource Was Not Available To The Client. The error from the *Stop-Service* cmdlet is Cannot Open XXX Service On Computer. The *Get-VM* cmdlet simply returns no information and no error. Therefore, check for console rights as a first step in troubleshooting.**

Launching Windows PowerShell with administrator rights

To perform tasks that require administrator rights, you must start the Windows PowerShell console with administrator rights. To do this, right-click the Windows PowerShell icon (the one pinned to the taskbar, the one on the Start window, or the one found by using the Search box in the Start window) and select the Run As Administrator option from the Action menu. The great advantage of this technique is that you can launch either the Windows PowerShell console (the first item on the menu) as an administrator, or from the same screen you can launch the Windows PowerShell ISE as an administrator. Figure 1-3 shows these options.

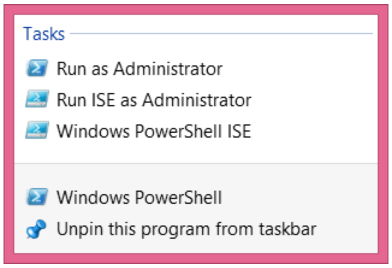


FIGURE 1-3 Right-click the Windows PowerShell icon to bring up the option to Run as Administrator.

Once you launch the Windows PowerShell console with administrator rights, the User Account Control (UAC) dialog box appears, requesting permission to allow Windows PowerShell to make changes to the computer. In reality, Windows PowerShell is not making changes to the computer, at least not yet. But using Windows PowerShell, you can certainly make changes to the computer if you have the rights. This is what the dialog box is prompting you for.

NOTE **It is possible to avoid this prompt by turning off UAC. However, UAC is a significant security feature, so I do not recommend disabling it. The UAC has been fine-tuned on Windows 7 and Windows 8. The number of UAC prompts has been greatly reduced from the number that used to exist with the introduction of UAC on Windows Vista.**

Now that you are running Windows PowerShell with administrator rights, you can do anything your account has permission to do. For example, if you run the *Get-Disk* cmdlets, you will see information similar to the following:

```
PS C:\> get-disk

Number Friendly Name                              Operational   Total Size Partition
                                                  status                   Style
------ -------------                              ------------  ---------- -----------
0      INTEL SSDSA2BW160G3L                       Online         149.05 GB MBR
```

Using Windows PowerShell cmdlets

Windows PowerShell cmdlets all work in a similar fashion. This simplifies their use. All Windows PowerShell cmdlets have a two-part name. The first part is a verb, although the verb is not always strictly grammatical. The verb indicates the action for the command to take. Examples of verbs include *Get*, *Set*, *Add*, *Remove*, and *Format*. The noun is the thing to which the action will apply. Examples of nouns include *Process*, *Service*, *Disk*, and *NetAdapter*. A dash combines the verb with the noun to complete the Windows PowerShell command. Windows PowerShell commands are named cmdlets (pronounced *command let*) because they behave like small commands or programs that are used standalone or pieced together through a mechanism called the *pipeline*. For more information about the pipeline, see Chapter 2, "Using Windows PowerShell Cmdlets."

The most common verb: *Get*

Out of nearly 2,000 cmdlets (and functions) on Windows 8, over 25 percent of them use the verb *Get*. The verb *Get* retrieves information. The noun portion of the cmdlet specifies the information retrieved. To obtain information about the processes on your system, open the Windows PowerShell console by either clicking the Windows PowerShell icon on the taskbar or typing **PowerShell** on the Start window of Windows 8 to bring up the search results for Windows PowerShell, as discussed in a preceding section, "Launching Windows PowerShell with administrator rights."

Once the Windows PowerShell console appears, run the *Get-Process* cmdlet. To do this, use the Windows PowerShell Tab Completion feature to complete the cmdlet name. Once the cmdlet name appears, press the Enter key to cause the command to execute.

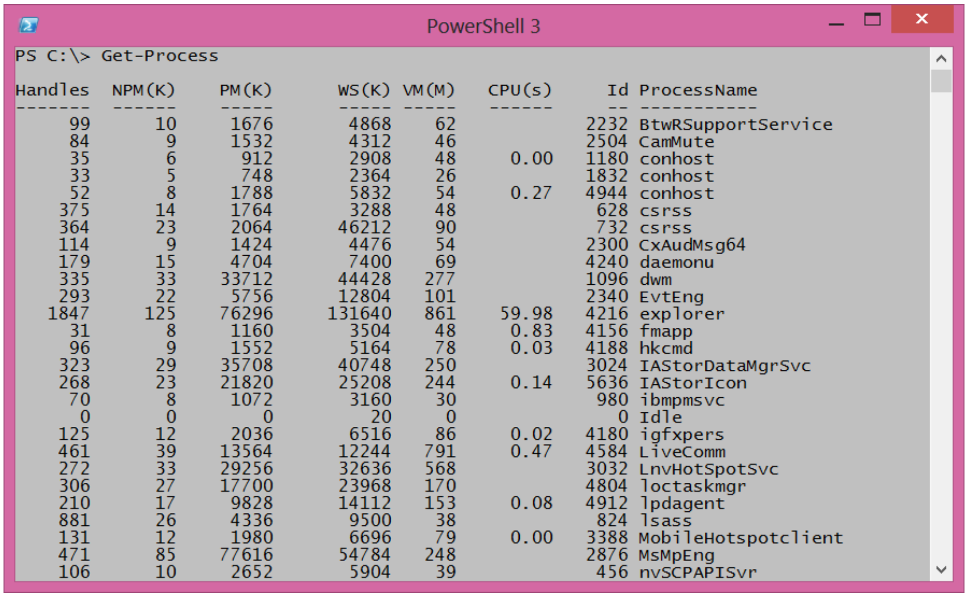
NOTE **The Windows PowerShell Tab Completion feature is a great time saver. It not only saves time by reducing the need for typing, but it also helps to ensure accuracy because tab completion accurately resolves cmdlet names. It is like a spelling checker for cmdlet names. For example, attempting to type a lengthy cmdlet name such as *Get-NetAdapter-EncapsulatedPacketTaskOffload* accurately could be an exercise in frustration. But if you use tab completion, you have to type only *Get-Net* and press the Tab key about six times before the correctly spelled cmdlet name appears in the Windows PowerShell console. Learning how to quickly and efficiently use tab completion is one of the keys to success for using Windows PowerShell.**

Finding process information

To use the Windows PowerShell Tab Completion feature to enter the *Get-Process* cmdlet name at the Windows PowerShell console command prompt, type the following on the first line of the Windows PowerShell console, then press the Tab key followed by Enter:

```
Get-Pro
```

This order of commands—command followed by Tab and Enter—is called *tab expansion*. Figure 1-4 shows the *Get-Process* command and associated output.



```
PS C:\> Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     99      10     1676       4868    62             2232 BtwRSupportService
     84       9     1532       4312    46             2504 CamMute
     35       6      912       2908    48     0.00   1180 conhost
     33       5      748       2364    26             1832 conhost
     52       8     1788       5832    54     0.27   4944 conhost
    375      14     1764       3288    48              628 csrss
    364      23     2064      46212    90              732 csrss
    114       9     1424       4476    54             2300 CxAudMsg64
    179      15     4704       7400    69             4240 daemonu
    335      33    33712      44428   277             1096 dwm
    293      22     5756      12804   101             2340 EvtEng
   1847     125    76296     131640   861    59.98   4216 explorer
     31       8     1160       3504    48     0.83   4156 fmapp
     96       9     1552       5164    78     0.03   4188 hkcmd
    323      29    35708      40748   250             3024 IAStorDataMgrSvc
    268      23    21820      25208   244     0.14   5636 IAStorIcon
     70       8     1072       3160    30              980 ibmpmsvc
      0       0        0         20     0                0 Idle
    125      12     2036       6516    86     0.02   4180 igfxpers
    461      39    13564      12244   791     0.47   4584 LiveComm
    272      33    29256      32636   568             3032 LnvHotSpotSvc
    306      27    17700      23968   170             4804 loctaskmgr
    210      17     9828      14112   153     0.08   4912 lpdagent
    881      26     4336       9500    38              824 lsass
    131      12     1980       6696    79     0.00   3388 MobileHotspotclient
    471      85    77616      54784   248             2876 MsMpEng
    106      10     2652       5904    39              456 nvSCPAPISvr
```

FIGURE 1-4 The Windows PowerShell *Get-Process* cmdlet returns detailed Windows process information.

To find information about Windows services, use the verb *Get* and the noun *Service*. In the Windows PowerShell console, type the following, then press the Tab key followed by Enter:

```
Get-Servi
```

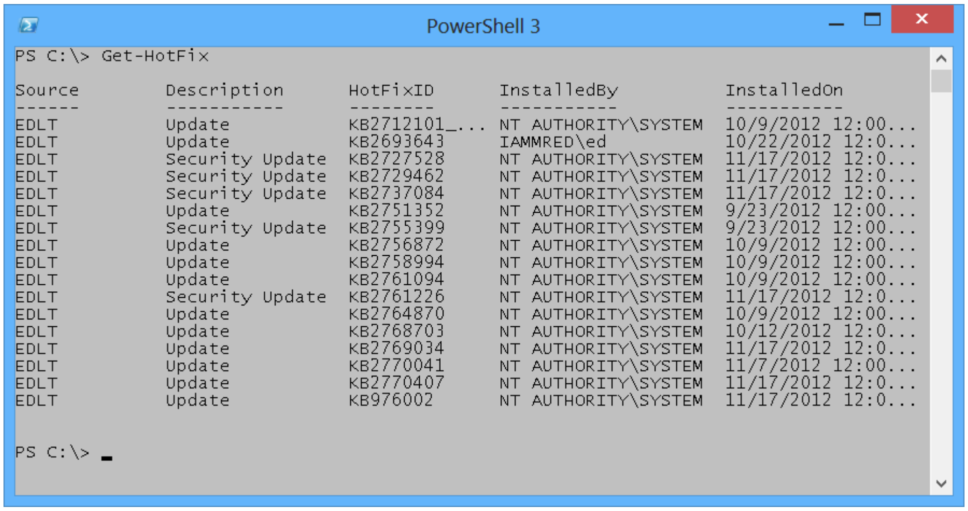
NOTE **It is a Windows PowerShell convention to use singular nouns. While not universally applied (my computer has about 50 plural nouns), it is a good place to start. So if you are not sure if a noun (or parameter) is singular or plural, choose the singular. Most of the time you will be correct.**

Identifying installed Windows hotfixes

To find a listing of Windows hotfixes applied to the current Windows installation, use the *Get-Hotfix* cmdlet. The verb is *Get* and the noun is *Hotfix*. In the Windows PowerShell console, type the following, then press the Tab key followed by Enter:

```
Get-Hotf
```

Figure 1-5 shows the *Get-Hotfix* command and associated output.



```
PS C:\> Get-HotFix

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
EDLT          Update           KB2712101_... NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2693643     IAMMRED\ed           10/22/2012 12:0...
EDLT          Security Update  KB2727528     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Security Update  KB2729462     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Security Update  KB2737084     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2751352     NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT          Security Update  KB2755399     NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT          Update           KB2756872     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2758994     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2761094     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Security Update  KB2761226     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2764870     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2768703     NT AUTHORITY\SYSTEM  10/12/2012 12:0...
EDLT          Update           KB2769034     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2770041     NT AUTHORITY\SYSTEM  11/7/2012 12:00...
EDLT          Update           KB2770407     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB976002      NT AUTHORITY\SYSTEM  11/17/2012 12:0...


PS C:\> _
```

FIGURE 1-5 Use the *Get-Hotfix* cmdlet to obtain a detailed listing of all applied Windows hotfixes.

Getting detailed service information

To find information about services on the system, use the *Get-Service* cmdlet. Once again, it is not necessary to type the entire command. The following command uses tab expansion to complete the *Get-Service* command and execute it:

```
Get-Servi
```

NOTE **The efficiency of tab expansion depends upon the number of cmdlets, functions, or modules installed on the computer. As more commands become available, the efficiency of tab expansion reduces correspondingly.**

The following (truncated) output appears following the *Get-Service* cmdlet:

```
PS C:\> Get-Service

Status   Name               DisplayName
------   ----               -----------
Running  AdobeActiveFile... Adobe Active File Monitor V6
Stopped  AeLookupSvc        Application Experience
Stopped  ALG                Application Layer Gateway Service
Stopped  AllUserInstallA... Windows All-User Install Agent
<TRUNCATED OUTPUT>
```

Identifying installed network adapters

To find information about network adapters on your Windows 8 or Windows Server 2012 machine, use the *Get-NetAdapter* cmdlet. Using tab expansion, type the following then press Tab, followed by Enter:

```
Get-NetA
```

The following example shows the command and associated output:

```
PS C:\> Get-NetAdapter

Name                      InterfaceDescription                    ifIndex Status
----                      --------------------                    ------- ------
Network Bridge            Microsoft Network Adapter Multiplexo...      29 Up
Ethernet                  Intel(R) 82579LM Gigabit Network Con...      13 Not Pre...
vEthernet (WirelessSwi... Hyper-V Virtual Ethernet Adapter #4          31 Up
vEthernet (External Sw... Hyper-V Virtual Ethernet Adapter #3          23 Not Pre...
vEthernet (InternalSwi... Hyper-V Virtual Ethernet Adapter #2          19 Up
Bluetooth Network Conn... Bluetooth Device (Personal Area Netw...      15 Disconn...
Wi-Fi                     Intel(R) Centrino(R) Ultimate-N 6300...      12 Up
```

Retrieving detected network connection profiles

If you want to see the network connection profile that Windows 8 or Windows Server 2012 detected for each interface, use the *Get-NetConnectionProfile* cmdlet. To run this command, use the following command with tab expansion:

```
Get-NetC
```

The following example shows the command and associated output:

```
PS C:\> Get-NetConnectionProfile


Name             : Unidentified network
InterfaceAlias   : vEthernet (InternalSwitch)
InterfaceIndex   : 19
NetworkCategory  : Public
IPv4Connectivity : NoTraffic
IPv6Connectivity : NoTraffic

Name             : Network  10
InterfaceAlias   : vEthernet (WirelessSwitch)
InterfaceIndex   : 31
NetworkCategory  : Public
IPv4Connectivity : Internet
IPv6Connectivity : NoTraffic
```

NOTE **Windows PowerShell is not case sensitive. There are a few instances where case sensitivity is an issue (for example, when using regular expressions) but cmdlet names, parameters, and values are not case sensitive. Windows PowerShell convention uses a combination of uppercase and lowercase letters, generally at syllable breaks in long noun names such as *NetConnectionProfile*. However, this is not a requirement for Windows PowerShell to interpret the command accurately. This combination of uppercase and lowercase letters is for readability. If you use tab expansion, Windows PowerShell automatically converts the commands to this format.**

Getting the current culture settings

A typical Windows computer has two categories of culture settings. The first category contains the culture settings for the current culture settings, which includes information about the keyboard layout and the display format of items such as numbers, currency, and dates. To find the value of these cultural settings, use the *Get-Culture* cmdlet. To call the *Get-Culture* cmdlet using tab expansion to complete the command, type the following at the command prompt of the Windows PowerShell console, then press the Tab key followed by Enter:

```
Get-Cu
```

When the command runs basic information such as the Language Code ID number (LCID), the name of the culture settings, in addition to the display name of the culture settings, return to the Windows PowerShell console. The following example shows the command and associated output:

```
PS C:\> Get-Culture

LCID             Name             DisplayName
----             ----             -----------
1033             en-US            English (United States)
```

The second category is the current user interface (UI) settings for Windows. The UI culture settings determine which text strings appear in user interface elements such as menus and error messages. To determine the current UI culture settings that are active, use the *Get-UI-Culture* cmdlet. Using tab expansion to call the *Get-UICulture* cmdlet, type the following, then press the Tab key followed by Enter:

```
Get-Ui
```

The following example shows the command and associated output:

```
PS C:\> Get-UICulture

LCID             Name             DisplayName
----             ----             -----------
1033             en-US            English (United States)
```

NOTE **On my computer, both the current culture and the current UI culture are the same. This is not always the case, and at times I have seen a computer have issues when the user interface is set for a localized language while the computer itself is set for U.S. English. This is especially problematic when using virtual machines created in other countries. In this case, even a simple task such as typing in a password becomes very frustrating. To fix these types of situations, you can use the *Set-Culture* cmdlet.**

Finding the current date and time

To find the current date or time on the local computer, use the *Get-Date* cmdlet. Tab expansion does not help much for this cmdlet because there are 15 cmdlets (on my computer) that have a cmdlet name that begins with the letters *Get-Da*. This includes all the Direct Access cmdlets as well as the Remote Access cmdlets. Therefore, using tab expansion to get the date requires me to type the following before pressing the Tab key followed by the Enter key:

```
Get-Dat
```

The preceding command syntax is the same number of keys to press as the following combined with the Enter key:

```
Get-Date
```

The following example shows the command and associated output:

```
PS C:\> Get-Date

Tuesday, November 20, 2012 9:54:21 AM
```

Generating a random number

Windows PowerShell 2.0 introduced the *Get-Random* cmdlet, and when I saw it I was not too impressed at first because I already knew how to generate a random number. As shown in the following example, I can use the .NET Framework *System.Random* class to create a new instance of the *System.Random* object and call the next method:

```
PS C:\> (New-Object system.random).next()
225513766
```

Needless to say, I did not create many random numbers. Who wants to do all that typing? But once I had the *Get-Random* cmdlet, I actually began using random numbers for all sorts of actions. For example, I have used the *Get-Random* cmdlet to do the following:

- Pick prize winners for the Scripting Games.
- Pick prize winners for Windows PowerShell user group meetings.
- Connect to remote servers in a random way for load-balancing purposes.
- Create random folder names.
- Create temporary users in Active Directory with random names.
- Wait a random amount of time prior to starting or stopping processes and services (great for performance testing).

The *Get-Random* cmdlet has turned out to be one of the more useful cmdlets. To generate a random number in the Windows PowerShell console using tab expansion, type the following on the first line in the console, then press the Tab key followed by the Enter key:

```
Get-R
```

The following example shows the command and associated output:

```
PS C:\> Get-Random
248797593
```

Supplying options for cmdlets

The easiest Windows PowerShell cmdlets to use require no options. Unfortunately, that is only a fraction of the total number of cmdlets (and functions) available in Windows PowerShell 3.0 as it exists on either Windows 8 or Windows Server 2012. Fortunately, the same tab expansion technique used to create the cmdlet names on the Windows PowerShell console works with parameters as well.

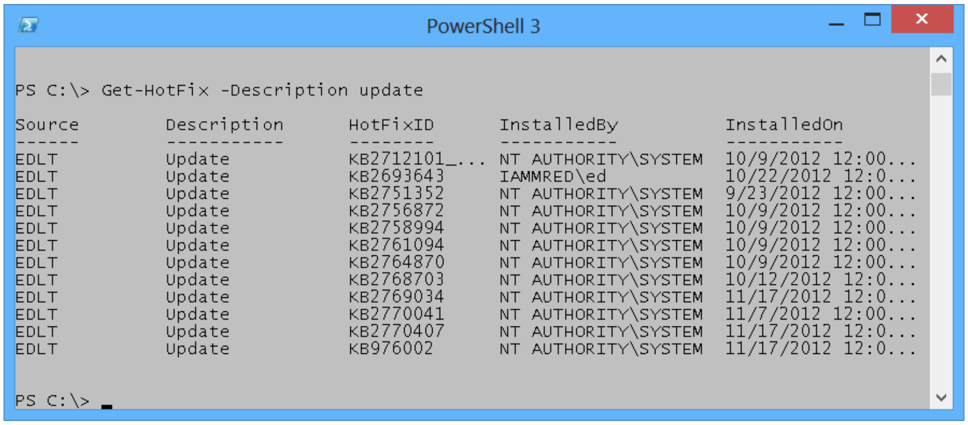
Using single parameters

When working with Windows PowerShell cmdlets, often the cmdlet requires only a single parameter to filter out the results. If a parameter is the default parameter, you do not have to specify the parameter name; you can use the parameter positionally. This means that the first value appearing after the cmdlet name is assumed to be a value for the default (or position 1) parameter. On the other hand, if a parameter is a named parameter, the parameter name (or parameter alias or partial parameter name) is always required when using the parameter.

Finding specific types of hotfixes

To find all the Windows Update hotfixes, use the *Get-HotFix* cmdlet with the -Description parameter and supply a value of update to the -Description parameter. This is actually easier than it sounds. Once you type **Get-Hot** and press the Tab key, you have the *Get-Hotfix* portion of the command. Then a space and *-D* + Tab completes the *Get-HotFix -Description* portion of the command. Now you need to type **Update** and press Enter. With a little practice, using tab expansion becomes second nature.

Figure 1-6 shows the *Get-Hotfix* command and associated output.



```
PS C:\> Get-HotFix -Description update

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
EDLT          Update           KB2712101_... NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2693643     IAMMRED\ed           10/22/2012 12:0...
EDLT          Update           KB2751352     NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT          Update           KB2756872     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2758994     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2761094     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2764870     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2768703     NT AUTHORITY\SYSTEM  10/12/2012 12:0...
EDLT          Update           KB2769034     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2770041     NT AUTHORITY\SYSTEM  11/7/2012 12:00...
EDLT          Update           KB2770407     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB976002      NT AUTHORITY\SYSTEM  11/17/2012 12:0...

PS C:\> _
```

FIGURE 1-6 Add the -Description parameter to the *Get-HotFix* cmdlet to see specific hotfixes such as updates in a filtered list.

If you attempt to find only update types of hotfixes by supplying the value update in the first position, an error appears. The following example shows the offending command and associated error:

```
PS C:\> Get-HotFix update
Get-HotFix : Cannot find the requested hotfix on the 'localhost' computer. Verify
the input and run the command again.
At line:1 char:1
+ Get-HotFix update
+ ~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (:) [Get-HotFix], ArgumentException
    + FullyQualifiedErrorId : GetHotFixNoEntriesFound,Microsoft.PowerShell.Commands
   .GetHotFixCommand
```

The error, while not really clear, seems to indicate that the *Get-HotFix* cmdlet attempts to find a hotfix named update. This is, in fact, the attempted behavior. The Help file information for the *Get-HotFix* cmdlet reveals that *-ID* is position 1, as shown in the following example:

```
    -Id <String[]>
        Gets only hotfixes with the specified hotfix IDs. The default is all
        hotfixes on the computer.

        Required?                    false
        Position?                    1
        Default value                All hotfixes
        Accept pipeline input?       false
        Accept wildcard characters?  False
```

You might ask, "What about using the -Description parameter?" The Help file states that the -Description parameter is a named parameter. This means you can use the -Description parameter only if you specify the parameter name, as shown earlier in this section. Following is the applicable portion of the Help file for the -Description parameter:

```
    -Description <String[]>
        Gets only hotfixes with the specified descriptions. Wildcards are
        permitted. The default is all hotfixes on the computer.

        Required?                    false
        Position?                    named
        Default value                All hotfixes
        Accept pipeline input?       false
        Accept wildcard characters?  True
```

Finding specific processes

To find process information about a single process, I use the -Name parameter. Because the -Name parameter is the default (position 1) parameter for the *Get-Process* cmdlet, you do not have to specify the -Name parameter when calling *Get-Process* if you do not want to do so. For example, to find information about the Windows PowerShell process by using the *Get-Process* cmdlet, perform the following command at the command prompt of the Windows PowerShell console by using tab expansion:

```
Get-Pro + <TAB> + <SPACE> + Po + <TAB> + <ENTER>
```

The following example shows the command and associated output:

```
PS C:\> Get-Process powershell

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    607      39   144552     164652   718     5.58   4860 powershell
```

You can tell the *Get-Process* cmdlet accepts the -Name parameter in a positional manner because the Help file states it is in position 1. The following example shows this position:

```
-Name <String[]>
    Specifies one or more processes by process name. You can type multiple
    process names (separated by commas) and use wildcard characters. The
    parameter name ("Name") is optional.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       true (ByPropertyName)
    Accept wildcard characters?  True
```

NOTE **Be careful using positional parameters because they can be confusing. For example, the first parameter for the *Get-Process* cmdlet is the *–Name* parameter, but the first position parameter for the *Stop-Parameter* is the *–ID* parameter. As a best practice, always refer to the Help files to see what the parameters actually are called and the position in which they are expected. This is even more important when using a cmdlet with multiple parameters, such as the *Get-Random* cmdlet discussed in the following section.**

Generating random numbers in a range

When used without any parameters, the *Get-Random* cmdlet returns a number that is in the range of 0 to 2,147,483,647. We have never had a Windows PowerShell user group meeting in which there were either 0 people in attendance or 2,147,483,647 people in attendance. Therefore, if you use the *Get-Random* cmdlet to select winners so you can hand out prizes at the end of the day, it is important to set a different minimum and maximum number.

NOTE **When you use the -Maximum parameter for the *Get-Random* cmdlet, keep in mind that the maximum number never appears. Therefore, if you have 15 people attending your Windows PowerShell user group meeting, you should set the -Maximum parameter to 16 (unless you do not like the number 15 person and do not want him to win any prizes).**

The default parameter for the *Get-Random* cmdlet is the -Maximum parameter. This means you can use the *Get-Random* cmdlet to generate a random number in the range of 0 to 20 by using tab expansion on the first line of the Windows PowerShell console. Remember that *Get-Random* never reaches the maximum number, so always use a number that is 1 greater than the desired upper number. Perform the following:

```
Get-R + <TAB> + <SPACE> + 21
```

If you want to generate a random number between 1 and 20, you might think you could use *Get-Random 1 21*, but that generates an error. The following example shows the command and error:

```
PS C:\> Get-Random 1 21
Get-Random : A positional parameter cannot be found that accepts argument ‘21’.
At line:1 char:1
+ Get-Random 1 21
+ ~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-Random], ParameterBindingEx
   ception
    + FullyQualifiedErrorId : PositionalParameterNotFound,Microsoft.PowerShell.Comm
   ands.GetRandomCommand
```

The error states that a positional parameter cannot be found that accepts argument 21. This is because *Get-Random* has only one positional parameter, the -Maximum parameter. The -Minimum parameter is a named parameter. This parameter appears in the Help file for the *Get-Random* cmdlet. I show you how to use the Help files in Chapter 2, “Using Windows PowerShell cmdlets.”

To generate a random number in the range of 1 to 20, use named parameters. For assistance in creating the command, use tab expansion for the cmdlet name as well as for the parameter names. Perform the following at the command prompt to create the command using tab expansion:

```
Get-R + <TAB> + -M + <TAB> + <SPACE> + 21 + -M + <TAB> + <SPACE> + 1 + <ENTER>
```

The following example shows the command and associated output:

```
PS C:\> Get-Random -Maximum 21 -Minimum 1
19
```

Introduction to parameter sets

One potentially confusing characteristic of Windows PowerShell cmdlets is that there are often different ways of using the same cmdlet. For example, you can specify the -Minimum and -Maximum parameters, but you cannot also specify the -Count parameter. This is a bit unfortunate because it would seem that using the -Minimum and -Maximum parameters to specify the minimum and maximum numbers for random numbers makes sense. When the Windows PowerShell user group has five prizes to give away, it is inefficient to write a script to generate the five random numbers or run the same command five times.

This is where command sets come into play. The -Minimum and -Maximum parameters specify the range within which to pick a single random number. To generate more than one random number, use the -Count parameter. The following example shows the two parameter sets:

```
Get-Random [[-Maximum] <Object>] [-Minimum <Object>] [-SetSeed <Int32>]
[<CommonParameters>]

Get-Random [-InputObject] <Object[]> [-Count <Int32>] [-SetSeed <Int32>]
[<CommonParameters>]
```

The first parameter set accepts -Maximum, -Minimum, and -SetSeed. The second parameter set accepts -InputObject, -Count, and -SetSeed. Therefore, you cannot use -Count with -Minimum or -Maximum because they are in two different groups of parameters (called parameter sets).

NOTE **It is quite common for Windows PowerShell cmdlets to have multiple parameter sets. Tab expansion offers only parameters from one parameter set. Therefore, when you choose a parameter such as –Count from *Get-Random*, the non-compatible parameters do not appear in tab expansion. This feature keeps you from creating invalid commands. For an overview of cmdlets parameter sets, use the *Get-Help* cmdlet.**

Generating a certain number of random numbers

The *Get-Random* cmdlet, when used with the -Count parameter, accepts an -InputObject parameter. The -InputObject parameter is quite powerful. The following excerpt from the Help file states that it accepts a collection of objects:

```
-InputObject <Object[]>
    Specifies a collection of objects. Get-Random gets randomly selected
    objects in random order from the collection. Enter the objects, a variable
    that contains the objects, or a command or expression that gets the
    objects. You can also pipe a collection of objects to Get-Random.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true (ByValue)
    Accept wildcard characters?  False
```

An array (or a range) of numbers just happens to also be a collection of objects. The easiest way to generate a range (or an array) of numbers is to use the *range operator*. The range operator is two dots (periods) between two numbers. As shown in the following example, the range operator does not require spaces between the numbers and dots:

```
PS C:\> 1..5
1
2
3
4
5
```

Now, to pick five random numbers from the range of 1 to 10 requires only the command to appear here. The parentheses are required around the range of 1 to 10 numbers to ensure the range of numbers is created prior to selecting five from the collection:

```
Get-Random -InputObject (1..10) -Count 5
```

The following example shows the command and associated output:

```
PS C:\> Get-Random -InputObject (1..10) -Count 5
7
5
10
1
8
```

Using command-line utilities

As easy as Windows PowerShell is to use, there are times when it is easier to find information by using a command-line utility. For example, to find IP configuration information, you need only use the Ipconfig.exe utility. You can type this directly into the Windows PowerShell console and read the output in the console. The following example shows the command and associated output in truncated form:

```
PS C:\> ipconfig

Windows IP Configuration

Wireless LAN adapter Local Area Connection* 14:

   Media State . . . . . . . . . . . : Media disconnected

   Connection-specific DNS Suffix  . :

Ethernet adapter vEthernet (WirelessSwitch):

   Connection-specific DNS Suffix  . : quadriga.com

   Link-local IPv6 Address . . . . . : fe80::915e:d324:aa0f:a54b%31

   IPv4 Address. . . . . . . . . . . : 192.168.13.220

   Subnet Mask . . . . . . . . . . . : 255.255.248.0

   Default Gateway . . . . . . . . . : 192.168.15.254

Wireless LAN adapter Local Area Connection* 12:

   Media State . . . . . . . . . . . : Media disconnected

   Connection-specific DNS Suffix  . :

Ethernet adapter vEthernet (InternalSwitch):

   Connection-specific DNS Suffix  . :

   Link-local IPv6 Address . . . . . : fe80::bd2d:5283:5572:5e77%19

   IPv4 Address. . . . . . . . . . . : 192.168.3.228
   Subnet Mask . . . . . . . . . . . : 255.255.255.0

   Default Gateway . . . . . . . . . : 192.168.3.100

<OUTPUT TRUNCATED>
```

To obtain the same information using Windows PowerShell, you need a more complex command. The command to obtain IP information is *Get-NetIPAddress*. But there are several advantages. For one thing, the output from the IpConfig.exe command is text, whereas the output from Windows PowerShell is an object. This means you can group, sort, filter, and format the output in an easy way.

The big benefit is that with the Windows PowerShell console, you have not only the simplicity of the command prompt, but you also have the powerful Windows PowerShell language built in. Therefore, if you need to refresh Group Policy three times and wait for five minutes between refreshes, you can use the command shown in the following example (looping is covered in Chapter 11, "Using Windows PowerShell Scripts"):

```
1..3 | % {gpupdate ; sleep 300}
```

Working with Help options

To use Help files effectively, the first thing you need to do is update them on your system. This is because Windows PowerShell 3.0 introduces a new model in which Help files update on a regular basis.

To update Help on your system, you must open the Windows PowerShell console with administrator rights. This is because Windows PowerShell Help files reside in the protected Windows\System32\WindowsPowerShell directory. Once you have launched the Windows PowerShell console with administrator rights, you need to ensure your computer has Internet access so it can download and install the updated files. If your computer does not have Internet connectivity, it will take several minutes before the command times out because Windows PowerShell tries really hard to obtain the updated files. If you run the *Update-Help* cmdlet with no parameters, Windows PowerShell attempts to download updated Help for all modules stored in the default Windows PowerShell modules locations that support updatable Help. To run *Update-Help* more than once a day, use the -Force parameter, as shown in the following example:

```
Update-Help -Force
```

Even without downloading updated Windows PowerShell Help, the Help subsystem displays the syntax of the cmdlet and other rudimentary information about the cmdlet.

To display Help information from the Internet, use the -*Online* switch. When used in this way, Windows PowerShell causes the default browser to open to the appropriate page from the Microsoft TechNet website.

In an enterprise, network administrators might want to use the *Save-Help* cmdlet to download Help from the Internet. Once downloaded, the *Update-Help* cmdlet can point to the network share for the files. This is an easy task to automate and can run as a scheduled task.

Summary

This chapter began with an overview of Windows PowerShell. In particular, it contrasted some of the differences and similarities between the Windows PowerShell console and the Windows PowerShell ISE. It explained that, regardless of where a Windows PowerShell command runs, the results are the same.

Windows PowerShell uses a verb and noun naming convention. To retrieve information, use the *Get* verb. To specify the type of information to obtain, use the appropriate noun. An example of this convention is the *Get-HotFix* cmdlet that returns hotfix information from the local system.

One of the most important concepts to understand about Windows PowerShell is that it allows a user to perform an action only if the security model permits it. For example, if a user has permission to stop a service by using the Services.MSC tool, the user will have permission to stop a service from within Windows PowerShell. But if a user is not permitted to stop a service elsewhere, Windows PowerShell does not permit the service to stop. Windows PowerShell also respects UAC. By default on Windows 7 and Windows 8, Windows PowerShell opens in least privilege mode. To perform actions requiring administrator rights, you must start Windows PowerShell as an administrator.

Many Windows PowerShell cmdlets run without any options and return valid data. This includes cmdlets such as *Get-Process* or *Get-Service*. However, most Windows PowerShell cmdlets require additional information to work properly. For example, the *Get-EventLog* cmdlet requires the name of a particular event log to return information.

The first thing you should do when logging onto the Windows PowerShell console is to run the *Update-Help* cmdlet. Note that this requires administrator rights and an Internet connection.

CHAPTER 7

Using Windows PowerShell remoting

- Using PowerShell remoting
- Configuring Windows PowerShell remoting
- Troubleshooting Windows PowerShell remoting

When you need to use Windows PowerShell on your local computer, it is pretty easy: You open the Windows PowerShell console or the Windows PowerShell ISE, and you run a command or a series of commands. Assuming you have rights to make the changes in the first place, it just works. But what if the change you need to make must be enacted on a hundred or a thousand computers? In the past, these types of changes required expensive specialized software packages, but with Windows PowerShell 3.0 running a command on a remote computer is as easy as running the command on your local computer; in some cases, it is even easier.

Using Windows PowerShell remoting

One of the great improvements in Windows PowerShell 3.0 is the change surrounding remoting. The configuration is easier than it was in Windows PowerShell 2.0, and in many cases, Windows PowerShell remoting "just works." When we talk about Windows PowerShell remoting, a bit of confusion can arise because there are several different ways of running commands against remote servers. Depending on your particular network configuration and security needs, one or more methods of remoting might not be appropriate.

Classic remoting

Classic remoting relies on protocols such as the Distributed Component Object Model (DCOM) and remote procedure call (RPC) to make connections to remote machines. Traditionally, these techniques require opening many ports in the firewall and starting various services the different cmdlets utilize. To find the Windows PowerShell cmdlets that natively support remoting, use the *Get-Help* cmdlet. Specify a value of computername for the parameter of the *Get-Help* cmdlet. This command produces a nice list of all cmdlets that have native support for remoting. The following example shows the command and associated output (this command does not display all cmdlets with support for computername unless the associated modules are preloaded):

```
PS C:\> Get-Help * -Parameter computername | sort name | ft name, synopsis -auto -wrap

Name                              Synopsis
----                              --------
Add-Computer                      Add the local computer to a domain or workgroup.
Add-Printer                       Adds a printer to the specified computer.
Add-PrinterDriver                 Installs a printer driver on the specified
                                  computer.
Add-PrinterPort                    Installs a printer port on the specified computer.
<…Output Truncated …>
```

Some of the cmdlets provide the ability to specify credentials. This allows you to use a different user account to make the connection and retrieve the data.

The following example shows this technique of using the computername and the credential parameters in a cmdlet:

```
PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName ex1 -Credential
nwtraders\administrator

TimeCreated           ProviderName                           Id Message
-----------           ------------                           -- -------
7/1/2012 11:54:14 AM MSExchange ADAccess                  2080 Process MAD.EXE (...
```

However, as mentioned earlier, use of these cmdlets often requires opening holes in the firewall or starting specific services. By default, these types of cmdlets fail when run against remote machines that have not relaxed access rules. The following example shows this type of error:

```
PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential
nwtraders\administrator
Get-WinEvent : The RPC server is unavailable
At linE:1 chaR:1
+ Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential iam
...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecifieD: (:) [Get-WinEvent], EventLogException
    + FullyQualifiedErrorId : System.Diagnostics.Eventing.Reader.EventLogException,
   Microsoft.PowerShell.Commands.GetWinEventCommand
```

Other cmdlets, such as *Get-Service* or *Get-Process,* do not have a credential parameter, and therefore the command impersonates the logged-on user, as shown in the following example:

```
PS C:\> Get-Service -ComputerName hyperv -Name bits

Status   Name               DisplayName
------   ----               -----------
Running  bits               Background Intelligent Transfer Ser...


PS C:\>
```

Just because the cmdlet does not support alternative credentials does not mean the cmdlet must impersonate the logged-on user. Holding down the Shift key and right-clicking on the Windows PowerShell icon brings up an action menu that allows you to run the program as a different user. When you use the Run as different user dialog box, you have alternative credentials available for Windows PowerShell cmdlets that do not support the credential parameter.

Configuring Windows PowerShell remoting

Windows Server 2012 installs with Windows Remote Management (WinRm) configured and running to support remote Windows PowerShell commands. WinRm is the Microsoft implementation of the industry standard WS-Management Protocol. As such, WinRM provides a firewall-friendly method of accessing remote systems in an interoperable manner. It is the remoting mechanism used by the new Common Information Model (CIM) cmdlets (the CIM cmdlets are covered in Chapter 9, "Using CIM"). As soon as Windows Server 2012 is up and running, you can make a remote connection and run commands or open an interactive Windows PowerShell console. A Windows 8 client, on the other hand, ships with WinRm locked down. Therefore, the first step is to use the *Enable-PSRemoting* function to configure remoting. When running the *Enable-PSRemoting* function, the following steps occur:

1. Starts or restarts the WinRM service.
2. Sets the WInRM service startup type to Automatic.
3. Creates a listener to accept requests from any Internet Protocol (IP) address.
4. Enables inbound firewall exceptions for *WS_Management* traffic.
5. Sets a target listener named *Microsoft.powershell*.
6. Sets a target listener named *Microsoft.powershell.workflow*.
7. Sets a target listener named *Microsoft.powershell32*.

During each step of this process, the function prompts you to agree or not agree to performing the specified action. If you are familiar with the steps the function performs, and you do not make any changes from the defaults, you can run the command with the Force switched parameter and it will not prompt prior to making the changes. The following example shows the syntax of this command:

```
Enable-PSRemoting -Force
```

The following example shows the use of the *Enable-PSRemoting* function in interactive mode, along with all associated output from the command:

```
PS C:\> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable remote management of this computer
by using the Windows Remote Management (WinRM) service.
 This includes:
    1. Starting or restarting (if already started) the WinRM service
    2. Setting the WinRM service startup type to Automatic
    3. Creating a listener to accept requests on any IP address
    4. Enabling Windows Firewall inbound rule exceptions for WS-Management traffic
(for http only).

Do you want to continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.

WinRM has been updated for remote management.
Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this mac
hine.
WinRM firewall exception enabled.


Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name:
microsoft.powershell SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will
allow selected users to remotely run Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name:
microsoft.powershell.workflow SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will
allow selected users to remotely run Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name:
microsoft.powershell32 SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will
allow selected users to remotely run Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y
PS C:\>
```

Once configured, use the *Test-WSMan* cmdlet to ensure the WinRM remoting is properly configured and is accepting requests. A properly configured system replies with the following data:

```
PS C:\> Test-WSMan -ComputerName w8c504


wsmid           : httP://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : httP://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

This cmdlet works with Windows PowerShell 2.0 remoting as well. Keep in mind that configuring WinRM through the *Enable-PSRemoting* function does not enable the WinRM firewall exception, and therefore PING commands will not work by default when pinging to a Windows 8 client system.

Running commands

For simple configuration on a single remote machine, entering a remote Windows PowerShell session is the answer. To enter a remote Windows PowerShell session, use the *Enter-PSSession* cmdlet to create an interactive remote Windows PowerShell session on a target machine. If you do not supply credentials, the remote session impersonates your current logon. The output in the following example illustrates connecting to a remote computer named dc1:

```
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> sl C:\
[dc1]: PS C:\> gwmi win32_bios


SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL   - 6



[dc1]: PS C:\> exit
PS C:\>
```

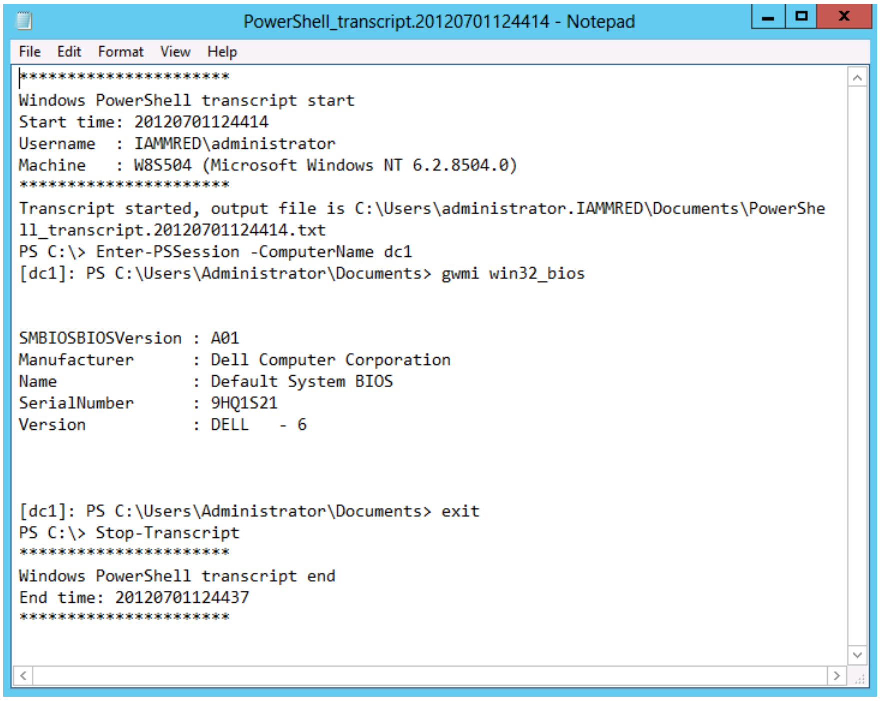
Once established, the Windows PowerShell prompt changes to include the name of the remote system. The *Set-Location* (*sl* is an alias) changes the working directory on the remote system to C:\. Next, the *Get-WmiObject* cmdlet retrieves the BIOS information on the remote system. The Exit command exits the remote session and the Windows PowerShell prompt returns to the default.

The good thing is that when using the Windows PowerShell transcript tool through *Start-Transcript*, the transcript tool captures output from the remote Windows PowerShell session as well as output from the local session. Indeed, all commands typed appear in the transcript.

The following commands illustrate beginning a transcript, entering a remote Windows PowerShell session, typing a command, exiting the session, and stopping the transcript:

```
PS C:\> Start-Transcript
Transcript started, output file is C:\Users\administrator.IAMMRED\Documents\PowerShe
ll_transcript.20120701124414.txt
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> gwmi win32_bios


SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL   - 6


[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Stop-Transcript
Transcript stopped, output file is C:\Users\administrator.IAMMRED\Documents\PowerShe
ll_transcript.20120701124414.txt
PS C:\>
```

Figure 7-1 shows the transcript from the preceding remote Windows PowerShell session. The transcript contains all commands, including the ones from the remote computer, and associated output.


PowerShell_transcript.20120701124414 - Notepad
File Edit Format View Help

```
**********************
Windows PowerShell transcript start
Start time: 20120701124414
Username  : IAMMRED\administrator
Machine   : W8S504 (Microsoft Windows NT 6.2.8504.0)
**********************
Transcript started, output file is C:\Users\administrator.IAMMRED\Documents\PowerShe
ll_transcript.20120701124414.txt
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> gwmi win32_bios


SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL   - 6



[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Stop-Transcript
**********************
Windows PowerShell transcript end
End time: 20120701124437
**********************
```

FIGURE 7-1 The transcript tool works in remote Windows PowerShell sessions as well as in local Windows PowerShell console sessions.

Running a single Windows PowerShell command

When you have a single command to run, it does not make sense to go through all the trouble of building and entering an interactive, remote Windows PowerShell session. Instead of creating a remote Windows PowerShell console session, you can run a single command by using the *Invoke-Command* cmdlet. If you have a single command to run, use the cmdlet directly and specify the computer name as well as any credentials required for the connection. The following example shows this technique, with the last process running on the ex1 remote server:

```
PS C:\> Invoke-Command -ComputerName ex1 -ScriptBlock {gps | select -Last 1}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName   PSComputerName
-------  ------    -----      ----- -----   ------     -- -----------   ------------
    224      34    47164      51080   532     0.58  10164 wsmprovhost   ex1
```

When you work interactively in a Windows PowerShell console, you might not want to type a long command, even when using tab expansion to complete the command. To shorten the amount of typing, you can use the *icm* alias for the *Invoke-Command* cmdlet. You can also rely upon positional parameters (the first parameter is the computer name and the second parameter is the script block). By using aliases and positional parameters, the previous command shortens considerably, as shown in the following example:

```
PS C:\> icm ex1 {gps | select -l 1}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName   PSComputerName
-------  ------    -----      ----- -----   ------     -- -----------   ------------
    221      34    47260      51048   536     0.33   4860 wsmprovhost   ex1
```

Running a single command against multiple computers

Use of the *Invoke-Command* exposes one of the more powerful aspects of Windows PowerShell remoting, which is running the same command against a large number of remote systems. The secret behind this power is that the computername parameter from the *Invoke-Command* cmdlet accepts an array of computer names. In the output appearing here, an array of computer names is stored in the variable *$cn*. Next, the *$cred* variable holds the *credential* object for the remote connections. Finally, the *Invoke-Command* cmdlet is used to make connections to all the remote machines and to return the BIOS information from the systems. The nice thing about this technique is that an additional parameter, PSComputerName, is added to the returning object, permitting easy identification of which BIOS is associated with which computer system. The following example shows the commands and associated output:

```
PS C:\> $cn = "dc1","dc3","ex1","sql1","wsus1","wds1","hyperv1","hyperv2","hyperv3"
PS C:\> $cred = Get-Credential iammred\administrator
PS C:\> Invoke-Command -cn $cn -cred $cred -ScriptBlock {gwmi win32_bios}
```

```
SMBIOSBIOSVersion : BAP6710H.86A.0072.2011.0927.1425
Manufacturer      : Intel Corp.
Name              : BIOS Date: 09/27/11 14:25:42 Ver: 04.06.04
SerialNumber      :
Version           : INTEL  - 1072009
PSComputerName    : hyperv3

SMBIOSBIOSVersion : A11
Manufacturer      : Dell Inc.
Name              : Phoenix ROM BIOS PLUS Version 1.10 A11
SerialNumber      : BDY91L1
Version           : DELL   - 15
PSComputerName    : hyperv2

SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL   - 6
PSComputerName    : dc1

SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32  Ver: 09.00.04
SerialNumber      : 3692-0963-1044-7503-9631-2546-83
Version           : VRTUAL - 3000919
PSComputerName    : wsus1

SMBIOSBIOSVersion : V1.6
Manufacturer      : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : To Be Filled By O.E.M.
Version           : 7583MS - 20091228
PSComputerName    : hyperv1

SMBIOSBIOSVersion : 080015
Manufacturer      : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : None
Version           : 091709 - 20090917
PSComputerName    : sql1

SMBIOSBIOSVersion : 080015
Manufacturer      : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : None
Version           : 091709 - 20090917
PSComputerName    : wds1

SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32  Ver: 09.00.04
SerialNumber      : 8994-9999-0865-2542-2186-8044-69
Version           : VRTUAL - 3000919
PSComputerName    : dc3
```

```
SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32  Ver: 09.00.04
SerialNumber      : 2301-9053-4386-9162-8072-5664-16
Version           : VRTUAL - 3000919
PSComputerName    : ex1



PS C:\>
```

Creating a persisted connection

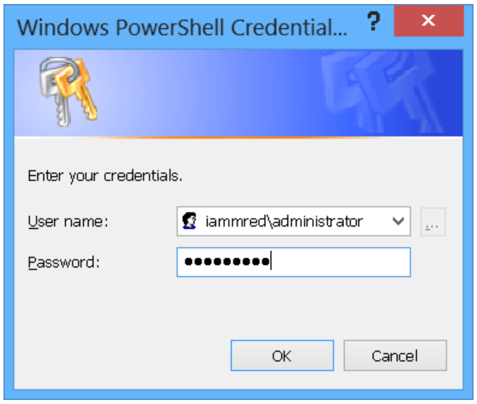
If you anticipate making multiple connections to a remote system, use the *New-PSSession* cmdlet to create a remote Windows PowerShell session. The *New-PSSession* cmdlet permits you to store the remote session in a variable and provides you with the ability to enter and leave the remote session as often as required, without the additional overhead of creating and destroying remote sessions. In the commands that follow, a new Windows PowerShell session is created through the *New-PSSession* cmdlet. The newly created session is stored in the *$dc1* variable. Next, the *Enter-PSSession* cmdlet is used to enter the remote session by using the stored session. A command retrieves the remote hostname, and the remote session is exited through the Exit command. Next, the session is re-entered and the last process retrieved. The session is exited once again. Finally, the *Get-PSSession* cmdlet retrieves Windows PowerShell sessions on the system, and all sessions are removed through the *Remove-PSSession* cmdlet:

```
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential iammred\administrator
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> hostname
dc1
[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> gps | select -Last 1

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    292       9    39536      50412   158     1.97   2332 wsmprovhost


[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Get-PSSession

 Id Name            ComputerName    State         ConfigurationName     Availability
 -- ----            ------------    -----         -----------------     ------------
  8 Session8        dc1             Opened        Microsoft.PowerShell     Available


PS C:\> Get-PSSession | Remove-PSSession
PS C:\>
```

If you have several commands, or if you anticipate making multiple connections, the *Invoke-Command* cmdlet accepts a session parameter in the same manner as the *Enter-PSSession* cmdlet does. In the output appearing here, a new *PSSession* is created to a remote computer named dc1. The remote session is used to retrieve two different pieces of information. Once completed, the session stored in the *$dc1* variable is explicitly removed:

```
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential iammred\administrator
PS C:\> Invoke-Command -Session $dc1 -ScriptBlock {hostname}
dc1
PS C:\> Invoke-Command -Session $dc1 -ScriptBlock {Get-EventLog application -Newest 1}

   Index Time          EntryType   Source                InstanceID Message PSCompu
                                                                            terName
   ----- ----          ---------   ------                ---------- ------- -------
   17702 Jul 01 12:59  Information ESENT                        701 DFSR... dc1


PS C:\> Remove-PSSession $dc1
```

You can also create persisted connection to multiple computers. This enables you to use the *Invoke-Command* cmdlet to run multiple commands against multiple remote computers. The first thing is to create a new *PSSession* that contains multiple computers. You can do this by using alternative credentials. Create a variable that holds the credential object returned by the *Get-Credential* cmdlet. A dialog box appears, permitting you to enter the credentials. Figure 7-2 shows the dialog box.


Windows PowerShell Credential...

Enter your credentials.

User name: iammred\administrator

Password: ●●●●●●●●●●

FIGURE 7-2 Store remote credentials in a variable populated through the *Get-Credential* cmdlet.

Once you have stored the credentials in a variable, create another variable to store the remote computer names. Next, use the *New-PSSession* cmdlet to create a new Windows PowerShell session using the computer names stored in the computer name variable and the credentials stored in the credential variable. To be able to reuse the Windows PowerShell remote session, store the newly created Windows PowerShell session in a variable as well. The following example illustrates storing the credentials, computer names, and newly created Windows PowerShell session:

```
$cred = Get-Credential -Credential iammred\administrator
$cn = "ex1","dc3"
$ps = New-PSSession -ComputerName $cn -Credential $cred
```

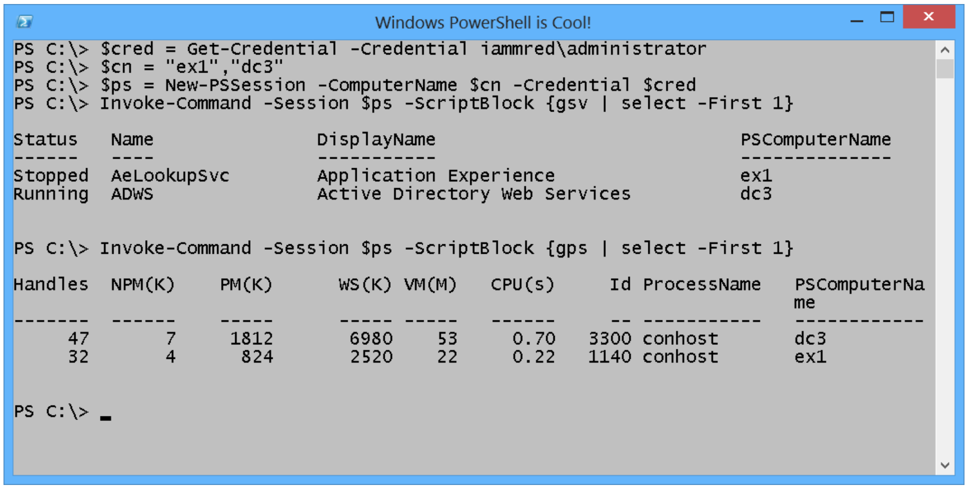
Once the Windows PowerShell session is created and stored in a variable, it can be used to execute commands against the remote computers. To do this, use the *Invoke-Command* cmdlet, as shown in the following example:

```
PS C:\> Invoke-Command -Session $ps -ScriptBlock {gsv | select -First 1}

Status   Name               DisplayName                           PSComputerName
------   ----               -----------                           --------------
Stopped  AeLookupSvc        Application Experience                ex1
Running  ADWS               Active Directory Web Services         dc3
```

The great thing about storing the remote connection in a variable is that it can be used for additional commands as well. The following example shows the command that returns the first process from each of the two remote computers:

```
PS C:\> Invoke-Command -Session $ps -ScriptBlock {gps | select -First 1}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName    PSComputerName
-------  ------    -----      ----- -----   ------     -- -----------    ------------
     47       7     1812       6980    53     0.70   3300 conhost        dc3
     32       4      824       2520    22     0.22   1140 conhost        ex1
```

Figure 7-3 shows the commands to store the credentials, create a remote Windows Power Shell connection to two different computers, and run two remote commands against them. Figure 7-3 also shows the output associated with the commands.



```
PS C:\> $cred = Get-Credential -Credential iammred\administrator
PS C:\> $cn = "ex1","dc3"
PS C:\> $ps = New-PSSession -ComputerName $cn -Credential $cred
PS C:\> Invoke-Command -Session $ps -ScriptBlock {gsv | select -First 1}

Status   Name               DisplayName                           PSComputerName
------   ----               -----------                           --------------
Stopped  AeLookupSvc        Application Experience                ex1
Running  ADWS               Active Directory Web Services         dc3


PS C:\> Invoke-Command -Session $ps -ScriptBlock {gps | select -First 1}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName    PSComputerNa
                                                                          me
-------  ------    -----      ----- -----   ------     -- -----------    ------------
     47       7     1812       6980    53     0.70   3300 conhost        dc3
     32       4      824       2520    22     0.22   1140 conhost        ex1


PS C:\> _
```

FIGURE 7-3 By creating and by storing a remote Windows PowerShell connection, it becomes easy to run commands against multiple computers.

Troubleshooting Windows PowerShell remoting

The first tool to use to see if Windows PowerShell remoting is working or not is the *Test-WSMan* cmdlet. Use it first on the local computer (no parameters are required). The following example shows the command and associated output:

```
PS C:\> Test-WSMan


wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

To test a remote computer, specify the -ComputerName parameter. The following example shows the command running against a Windows Server 2012 domain controller named dc3:

```
PS C:\> Test-WSMan -ComputerName dc3


wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

However, the *Test-WSMan* cmdlet also works against a computer running Windows PowerShell 2.0. The following example shows the command running against a Windows Server 2008 domain controller named dc1:

```
PS C:\> Test-WSMan -ComputerName dc1


wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 2.0
```

To examine a specific Windows PowerShell session, use the *Get-PSSession* cmdlet. The easiest way to do this is to pipeline the variable containing the Windows PowerShell session to the *Get-PSSession* cmdlet. The key items to pay attention to are the computer name, the state of the session, and the availability of the session. The following example shows this technique:

```
PS C:\> $ps | Get-PSSession

 Id Name            ComputerName    State         ConfigurationName     Availability
 -- ----            ------------    -----         -----------------     ------------
  3 Session3        ex1             Opened        Microsoft.PowerShell     Available
  4 Session4        dc3             Opened        Microsoft.PowerShell     Available
```

To focus on a specific session, reference the session by either ID or by Name. Send the returned session object over the pipeline to the *Format-List* cmdlet and select all the properties. The following example shows this technique (using *fl* as an alias for the *Format-List* cmdlet):

```
PS C:\> Get-PSSession -Name Session4 | fl *


State                  : Opened
IdleTimeout            : 7200000
OutputBufferingMode    : None
ComputerName           : dc3
ConfigurationName      : Microsoft.PowerShell
InstanceId             : c15cc80d-64f0-4096-a010-0211f0188aec
Id                     : 4
Name                   : Session4
Availability           : Available
ApplicationPrivateData : {PSVersionTable}
Runspace               : System.Management.Automation.RemoteRunspace
```

You can remove a remote Windows PowerShell session by pipelining the results of *Get-PSSession* to the *Remove-PSSession* cmdlet, as shown in the following example:

```
Get-PSSession -Name Session4 | Remove-PSSession
```

You can also remove a *PS* session directly by specifying the name to the *Remove-PSSession* cmdlet, as shown in the following example:

```
Remove-PSSession -Name session3
```

Summary

This chapter discussed the reason to use Windows PowerShell remoting. We covered the different types of remoting, such as classic remoting and Windows Remote Management Windows PowerShell (WinRM) remoting. In addition, we covered how to enable Windows PowerShell remoting and how to run a single command against a remote computer. Finally, we examined running multiple commands, creating persisted connections, and troubleshooting Windows PowerShell remoting.

Index