SAMPLE CHAPTER

Learn WINDOWS POWERSHELL IN A MONTH OF LUNCHES

DON JONES

MANNING

Learn Windows PowerShell in a Month of Lunches

by Don Jones

Chapter 10

brief contents

10 Remote control: one to one, and one to many

When I first started using PowerShell (in version 1), I was playing around with the `Get-Service` command, and noticed that it had a `-computerName` parameter. Hmmm ... does that mean it can get services from other computers, too? After a bit of experimenting, I discovered that's exactly what it did. I got very excited and started looking for `-computerName` parameters on other cmdlets, and was disappointed to find that there were very few. A few more were added in v2, but the commands that have this parameter are vastly outnumbered by the commands that don't.

What I've realized since is that PowerShell's creators are a bit lazy—and that's a good thing! They didn't want to have to code a `-computerName` parameter for every single cmdlet, so they created a shell-wide system called *remoting*. Basically, it enables any cmdlet to be run on a remote computer. In fact, you can even run commands that exist on the remote computer but that don't exist on your own computer—meaning that you don't always have to install every single administrative cmdlet on your workstation. This remoting system is powerful, and it offers a number of interesting administrative capabilities.

10.1 The idea behind remote PowerShell

Remote PowerShell works somewhat similarly to Telnet and other age-old remote control technologies. When you run a command, it's actually running *on* the remote computer. Only the results of that command come back to your computer. Rather than using Telnet or SSH, however, PowerShell uses a new communications protocol called Web Services for Management (WS-MAN).

WS-MAN operates entirely over HTTP or HTTPS, making it easy to route through firewalls if necessary (because each of those protocols uses a single port to communicate). Microsoft's implementation of WS-MAN comes in the form of a background service, Windows Remote Management (WinRM). WinRM is installed along with PowerShell v2 and is started by default on server operating systems like Windows Server 2008 R2. It's installed on Windows 7 by default, but the service is disabled.

You've already learned that Windows PowerShell cmdlets all produce objects as their output. When you run a remote command, its output objects need to be put into a form that can be easily transmitted over a network using the HTTP (or HTTPS) protocol. XML, it turns out, is an excellent way to do that, so PowerShell automatically *serializes* those output objects into XML. The XML is transmitted across the network and is then *deserialized* on your computer back into objects that you can work with inside PowerShell.

Why should you care how this output is returned? Because those serialized objects are really just snapshots, of sorts; they don't update themselves continually. For example, if you were to get the objects that represent the processes running on a remote computer, what you'd get back would only be accurate for the exact point in time at which those objects were generated. Values like memory usage and CPU utilization won't be updated to reflect subsequent conditions. In addition, you can't tell the deserialized objects to do anything—you can't instruct one to stop itself, for example.

Those are basic limitations of remoting, but they don't stop you from doing some pretty amazing stuff. In fact, you can tell a remote process to stop itself—you just have to be a bit clever about it. I'll show you how in a bit.

There are two basic requirements to make remoting work:

- Both your computer and the one you want to send commands to must be running Windows PowerShell v2. Windows XP is the oldest version of Windows on which you can install PowerShell v2, so it's the oldest version that can participate in remoting.
- Ideally, both computers need to be members of the same domain, or of trusted/trusting domains. It's possible to get remoting to work outside of a domain, but it's tricky, and I won't be covering it in this chapter. To learn more about that scenario, open PowerShell and run `Help about_remote_troubleshooting`.

TRY IT NOW I'm hoping that you'll be able to follow along with some of the examples in this chapter. To do so, you'll ideally have a second test computer (or virtual machine) that's in the same Active Directory domain as the test computer you've been using up to this point. That second computer can be running any version of Windows, provided PowerShell v2 is installed. If you can't set up an additional computer or virtual machine, use "localhost" to create remoting connections to your current computer. You're still using

remoting, but it isn't as exciting to be "remote controlling" the computer that you're sitting in front of.

10.2 WinRM overview

Let's talk a bit about WinRM, because you're going to have to configure it in order to start using remoting. Once again, you only need to configure WinRM—and PowerShell remoting—on those computers that will *receive* incoming commands. In most of the environments I've worked in, the administrators have enabled remoting on every Windows-based computer (keep in mind that PowerShell and remoting are supported all the way back to Windows XP). Doing so gives you the ability to remote into client desktop and laptop computers in the background (meaning the users of those computers won't know you're doing so), which can be tremendously useful.

WinRM isn't unique to PowerShell. In fact, it's likely that Microsoft will start using it for more and more administrative communications—even things that use other protocols today. With that in mind, Microsoft made WinRM able to route traffic to multiple administrative applications—not just PowerShell. WinRM essentially acts as a dispatcher: when traffic comes in, WinRM decides which application needs to deal with that traffic. All WinRM traffic is tagged with the name of a recipient application, and those applications must register with WinRM to listen for incoming traffic on their behalf. In other words, you'll not only need to enable WinRM, but you'll also need to tell PowerShell to register as an *endpoint* with WinRM.

One way to do that is to open a copy of PowerShell—making sure that you're running it as an Administrator—and run the `Enable-PSRemoting` cmdlet. You might sometimes see references to a different cmdlet, called `Set-WSManQuickConfig`. There's no need to run that one; `Enable-PSRemoting` will call it for you, and `Enable-PSRemoting` does a few extra steps that are necessary to get remoting up and running. All told, the cmdlet will start the WinRM service, configure it to start automatically, register PowerShell as an endpoint, and even set up a Windows Firewall exception to permit incoming WinRM traffic.

> **TRY IT NOW** Go ahead and enable remoting on your second computer (or on the first one, if that's the only one you have to work with). Make sure you're running PowerShell as an Administrator (it should say "Administrator" in the window's title bar). If you're not, close the shell, right-click the PowerShell icon in the Start menu, and select Run as Administrator from the context menu.

If you're not excited about having to run around to every computer to enable remoting, don't worry: you can also do it with a Group Policy object (GPO), too. The necessary GPO settings are built into Windows Server 2008 R2 domain controllers (and you can download an ADM template from download.Microsoft.com to add these GPO settings to an older domain's domain controllers). Just open a Group Policy object and look under the Computer Configuration, then under Administrative Templates, then

under Windows Components. Near the bottom of the list, you'll find both Remote Shell and Windows Remote Management. For now, I'm going to assume that you'll run `Enable-PSRemoting` on those computers that you want to configure, because at this point you're probably just playing around with a virtual machine or two.

> **NOTE** The about_remote_troubleshooting help topic in PowerShell provides more coverage on using GPOs. Look for the "How to enable remoting in an enterprise" and "How to enable listeners by using a Group Policy" sections within that help topic.

WinRM v2 (which is what PowerShell uses) defaults to using TCP port 5985 for HTTP and 5986 for HTTPS. Those ports help to ensure it won't conflict with any locally installed web servers, which tend to listen to 80 and 443 instead. You can configure WinRM to use alternative ports, but I don't recommend doing so. If you leave those ports alone, all of PowerShell's remoting commands will run normally. If you change the ports, you'll have to always specify an alternative port when you run a remoting command, which just means more typing for you.

If you absolutely must change the port, you can do so by running this command:

```
Winrm set winrm/config/listener?Address=*+Transport=HTTP
➥ @{Port="1234"}
```

In this example, "1234" is the port you want. Modify the command to use HTTPS instead of HTTP to set the new HTTPS port.

> **DON'T TRY IT NOW** Although you may want to change the port in your production environment, don't change it on your test computer. Leave WinRM using the default configuration so that the remainder of this book's examples will work for you without modification.

I should admit that there is a way to configure WinRM on client computers to use alternative default ports, so that you're not constantly having to specify an alternative port when you run commands. But for now let's stick with the defaults Microsoft came up with.

> **NOTE** If you do happen to browse around in the Group Policy object settings for Remote Shell, you'll notice that you can set things like how long a remoting session can sit idle before the server kills it, how many concurrent users can remote into a server at once, how much memory and how many processes each remote shell can utilize, and the maximum number of remote shells a given user can open at once. These are all great ways to help ensure that your servers don't get overly burdened by forgetful administrators! By default, however, you *do* have to be an Administrator to use remoting, so you don't need to worry about ordinary users clogging up your servers.

10.3 Using Enter-PSSession and Exit-PSSession for 1:1 remoting

PowerShell uses remoting in two distinct ways. The first is called *one-to-one,* or 1:1, remoting (the second way is one-to-many remoting, and you'll see it in the next section). With this kind of remoting, you're basically accessing a shell prompt on a single remote computer. Any commands you run will run directly on that computer, and you'll see results in the shell window. This is vaguely similar to using Remote Desktop Connection, except that you're limited to the command-line environment of Windows PowerShell. Oh, and this kind of remoting uses a *fraction* of the resources that Remote Desktop requires, so it imposes much less overhead on your servers!

To establish a one-to-one connection with a remote computer, run this command:

```
Enter-PSSession -computerName Server-R2
```

Of course, you'll need to provide the correct computer name instead of `Server-R2`.

Assuming you enabled remoting on that computer, that you're all in the same domain, and that your network is functioning correctly, you should get a connection going. PowerShell lets you know that you've succeeded by changing the shell prompt:

```
[server-r2] PS C:\>
```

That prompt tells you that everything you're doing is taking place on Server-R2 (or whatever server you connected to). You can run whatever commands you like. You can even import any modules, or add any PSSnapins, that happen to reside on that remote computer.

> **TRY IT NOW** Go ahead and try to create a remoting connection to your second computer or virtual machine. If you haven't yet done so, you'll need to enable remoting on that computer before you try to connect to it. Note that you're going to need to know the real computer name of the remote computer; WinRM won't, by default, permit you to connect by using its IP address or a DNS alias.

Even your permissions and privileges carry over across the remote connection. Your copy of PowerShell will pass along whatever security token it's running under (it does this with Kerberos, so it doesn't pass your username or password across the network). Any command you run on the remote computer will run under your credentials, so you'll be able to do anything you'd normally have permission to do. It's just like logging directly into that computer's console and using its copy of PowerShell directly.

Well, almost. There are a couple of differences:

- Even if you have a PowerShell profile script on the remote computer, it won't run when you connect using remoting. We haven't fully covered profile scripts yet (they're in chapter 24), but suffice to say that they're a batch of commands that run automatically each time you open the shell. Folks use them to

automatically load shell extensions and modules and so forth. That doesn't happen when you remote into a computer, so be aware of that.

- You're still restricted by the remote computer's execution policy. Let's say your local computer's policy is set to `RemoteSigned`, so that you can run local, unsigned scripts. That's great, but if the remote computer's policy is set to the default, `Restricted`, it won't be running any scripts for you when you're remoting into it.

Aside from those two fairly minor caveats, you should be good to go. Oh, wait—what do you do when you're done running commands on the remote computer? Many PowerShell cmdlets come in pairs, with one cmdlet doing something and the other doing the opposite. In this case, if `Enter-PSSession` gets you *into* the remote computer, can you guess what would get you *out* of the remote computer? If you guessed `Exit-PSSession`, give yourself a prize. The command doesn't need any parameters; just run it and your shell prompt will change back to normal, and the remote connection will close automatically.

TRY IT NOW Go ahead and exit the remoting session, if you created one. We're done with it for now.

What if you forget to run `Exit-PSSession` and instead close the PowerShell window? Don't worry. PowerShell and WinRM are smart enough to figure out what you did, and the remote connection will close all by itself.

I do have one caution to offer. When you're remoting into a computer, don't run `Enter-PSSession` *from that computer* unless you fully understand what you're doing. Let's say you work on Computer A, which runs Windows 7. You remote into Server-R2. Then, at the PowerShell prompt, you run this:

```
[server-r2] PS C:\>enter-pssession server-dc4
```

Now, Server-R2 is maintaining an open connection to Server-DC4. That can start to create a "remoting chain" that's hard to keep track of, and which imposes unnecessary overhead on your servers. There are times when you might *have* to do this—I'm thinking mainly of instances where a computer like Server-DC4 sits behind a firewall and you can't access it directly, so you use Server-R2 as a middleman to hop over to Server-DC4. But, as a general rule, try to avoid remote chaining.

When you're using this one-to-one remoting, you don't need to worry about objects being serialized and deserialized. As far as you're concerned, you're typing directly on the remote computer's console. If you retrieve a process and pipe it to `Stop-Process`, it'll stop as you would expect it to.

10.4 Using Invoke-Command for one-to-many remoting

The next trick—and honestly, this is one of the coolest things in Windows PowerShell—is to send a command to *multiple remote computers at the same time*. That's right, full-scale distributed computing. Each computer will independently execute the

command and send the results right back to you. It's all done with the `Invoke-Command` cmdlet, and it's called *one-to-many,* or 1:n, remoting.

The command looks something like this:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 |
➥ Where { $_.EventID -eq 1212 }}
```

TRY IT NOW Go ahead and run this command. Substitute the name of your remote computer (or computers) where I've put my three computer names.

Everything in those outermost curly braces, the `{}`, will get transmitted to the remote computers—all three of them. By default, PowerShell will talk to up to 32 computers at once; if you specified more than that, it will queue them up, so that as one computer completes, the next one in line will begin. If you have an awesome network and powerful computers, you could raise that number by specifying the `-throttleLimit` parameter of `Invoke-Command`—read the command's help for more information.

Be careful about the punctuation

We need to pause for a moment and dig into the preceding example command, because this is a case where PowerShell's punctuation can get confusing, and that confusion can make you do the wrong thing when you start constructing these command lines on your own.

There are two commands in that example that use curly braces: `Invoke-Command` and `Where` (which is an alias for `Where-Object`). `Where` is entirely nested within the outer set of braces. The outermost set of braces enclose everything that's being sent to the remote computers for execution:

```
Get-EventLog Security -newest 200 | Where { $_.EventID -eq 1212 }
```

It can be tough to follow that nesting of commands, especially in a book like this where the physical width of the page makes it necessary to display the command across several lines of text.

Don't read any further until you're sure you can identify the exact command that's being sent to the remote computer, and that you understand what each matched set of curly braces is for.

I should tell you that you won't see the `-command` parameter in the help for `Invoke-Command`—but the command I just showed you will work fine. The `-command` parameter is an *alias,* or nickname, for the `-scriptblock` parameter that you *will* see listed in the help. I have an easier time remembering `-command`, so I tend to use it instead of `-scriptblock`, but they both work the same way.

If you read the help for `Invoke-Command` carefully (see how I'm continuing to push those help files?), you'll also notice a parameter that lets you specify a script file, rather than a command. That parameter lets you send an entire script from your local

computer to the remote computers—meaning you can automate some pretty complex tasks and have each computer do its own share of the work.

> **TRY IT NOW** Make sure you can identify the `-scriptblock` parameter in the help for `Invoke-Command`, and that you can spot the parameter that would enable you to specify a file path and name instead of a script block.

I want to circle back to the `-computerName` parameter for a bit. When I first used `Invoke-Command`, I typed a comma-separated list of computer names, just as I did in the previous example. But I work with a *lot* of computers, so I didn't want to have to type them all in every time. I keep text files for some of my common computer categories, like web servers and domain controllers. Each text file contains one computer name per line, and that's it—no commas, no quotes, no nothing. PowerShell makes it easy for me to use those files:

```
Invoke-Command -command { dir }
➥ -computerName (Get-Content webservers.txt)
```

The parentheses here force PowerShell to execute `Get-Content` first—pretty much the same way parentheses work in math. The results of `Get-Command` are then stuck into the `-computerName` parameter, which then works against each of the computers that are listed in the file.

I also sometimes want to query computer names from Active Directory. This is a bit trickier. I can use the `Get-ADComputer` command (from the ActiveDirectory module in Windows Server 2008 R2) to retrieve computers, but I can't stick that command in parentheses like I did with `Get-Content`. Why not? Because `Get-Content` produces simple strings of text, which `-computerName` is expecting. `Get-ADComputer`, on the other hand, produces entire computer objects, and the `-computerName` parameter won't know what to do with them.

If I want to use `Get-ADComputer`, I need to find a way to get just the *values* from those computer objects' `Name` properties. Here's how:

```
Invoke-Command -command { dir } -computerName (
➥ Get-ADComputer -filter * -searchBase "ou=Sales,dc=company,dc=pri" |
➥ Select-Object -expand Name )
```

> **TRY IT NOW** If you're running PowerShell on a Windows Server 2008 R2 domain controller, or on a Windows 7 computer that has the Remote Server Administration Tools installed, you can run `Import-Module ActiveDirectory` and then try the preceding command. If your test domain doesn't have a Sales OU that contains a computer account, then change `ou=Sales` to `ou=Domain Controllers`, and be sure to change `company` and `pri` to the appropriate values for your domain (for example, if your domain is mycompany.org, you would substitute `mycompany` for `company` and `org` for `pri`).

Within the parentheses, I've piped the computer objects to `Select-Object`, and I've used its `-expand` parameter. I'm telling it to expand the `Name` property of whatever

came in—in this case, those computer objects. The result of that entire parenthetical expression will be a bunch of computer names, not computer objects—and computer names are exactly what the `-computerName` parameter wants to see.

Just to be complete, I should mention that the `-filter` parameter of `Get-ADComputer` specifies that all computers should be included in the command's output. The `-searchBase` parameter tells the command to start looking for computers in the specified location—in this case, the Sales OU of the company.pri domain. The `Get-ADComputer` command is only available on Windows Server 2008 R2, and on Windows 7 after installing the Remote Server Administration Tools (RSAT). On those operating systems, you have to run `Import-Module ActiveDirectory` to load the Active Directory cmdlets into the shell so that they can be used.

10.5 Differences between remote and local commands

I want to explain a bit about the differences between running commands using `Invoke-Command`, and running those same commands locally, as well as the differences between remoting and other forms of remote connectivity. For this entire discussion, I'll use this command as my example:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 |
➥ Where { $_.EventID -eq 1212 }}
```

Let's look at some alternatives, and why they're different.

10.5.1 Invoke-Command versus -ComputerName

Here's an alternative way to perform that same basic task:

```
Get-EventLog Security -newest 200
➥ -computerName Server-R2,Server-DC4,Server12
➥ | Where { $_.EventID -eq 1212 }
```

Here, I've used the `-computerName` parameter of `Get-EventLog`, rather than invoking the entire command remotely. I'll get more or less the same results, but there are some important differences in how the command executes:

- Using this command, the computers will be contacted sequentially rather than in parallel, which means the command may take longer to execute.
- The output won't include a `PSComputerName` property, which may make it harder for me to tell which result came from which computer.
- The connection won't be made using WinRM, but will instead use whatever underlying protocol the .NET Framework decides on. I don't know what that is, and it might be harder to get the connection through any firewalls that are between me and the remote computer.
- I'm querying 200 records from each of the three computers, and only then am I filtering through them to find the ones with `EventID 1212`. That means I am probably bringing over a lot of records that I don't want.

- I'm getting back event log objects that are fully functional.

These differences apply to any cmdlet that has a `-computerName` parameter. Generally speaking, it can be more efficient and effective to use `Invoke-Command` rather than a cmdlet's `-computerName` parameter.

Here's what would have happened if I'd used the original `Invoke-Command` instead:

- The computers would have been contacted in parallel, meaning the command could complete somewhat more quickly.
- The output would have included a `PSComputerName` property, enabling me to more easily distinguish the output from each computer.
- The connection would have been made through WinRM, which uses a single, predefined port that can be easier to get through any intervening firewalls.
- Each computer would have queried the 200 records and filtered them *locally*. The only data transmitted across the network would have been the result of that filtering, meaning that only the records I cared about would have been transmitted.
- Before transmitting, each computer would have serialized its output into XML. My computer would have received that XML and deserialized it back into something that looks like objects. But they wouldn't have been real event log objects, and that might limit what I could do with them once they were on my computer.

That last point is a big distinction between using a `-computerName` parameter and using `Invoke-Command`. Let's discuss that distinction.

10.5.2 Local versus remote processing

Here's my original example again:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 |
➥ Where { $_.EventID -eq 1212 }}
```

Now, compare it to this alternative:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 } |
➥ Where { $_.EventID -eq 1212 }
```

The differences are subtle. Actually, there's only one difference: I moved one of those curly braces.

In the second version, only `Get-EventLog` is being invoked remotely. All of the results generated by `Get-EventLog` will be serialized and sent to my computer, where they'll be deserialized into objects and then piped to `Where` and filtered. The second version of the command is less efficient, because a lot of unnecessary data is being transmitted across the network, and my one computer is having to filter the results from three computers, rather than those three computers filtering their own results for me. The second version, in other words, is a bad idea.

Let's look at two versions of another command. Here's the first:

```
Invoke-Command -computerName Server-R2
➥ -command { Get-Process -name Notepad } |
➥ Stop-Process
```

Here's the second version:

```
Invoke-Command -computerName Server-R2
➥ -command { Get-Process -name Notepad |
➥ Stop-Process }
```

Once again, the only difference between these two is the placement of a curly brace. In this example, however, the first version of the command won't work.

Look carefully: I'm sending `Get-Process -name Notepad` to the remote computer. The remote computer retrieves the specified process, serializes it into XML, and sends it to me across the network. My computer receives that XML, deserializes it back into an object, and pipes it to `Stop-Process`. The problem is that the deserialized XML doesn't contain enough information for my computer to realize that the process *came from a remote machine*. Instead, my computer will try to stop the Notepad process *running locally*, which isn't what I wanted at all.

The moral of the story is to always complete as much of your processing on the remote computer as possible. The only thing you should expect to do with the results of `Invoke-Command` is to display them or store them, as a report or data file or something. The second version of my command follows that advice: what's being sent to the remote computer is `Get-Process -name Notepad | Stop-Process`, so the entire command—both getting the process and stopping it—happens on the remote computer. Because `Stop-Process` doesn't normally produce any output, there won't be any objects to serialize and send to me, so I won't see anything on my local console. But the command will do what I want: stop the Notepad process *on the remote computer*, not on my local machine.

Whenever I use `Invoke-Command`, I always look at the commands after it. If I see commands for formatting, or for exporting data, I'm fine, because it's okay to do those things with the results of `Invoke-Command`. But if `Invoke-Command` is followed by action cmdlets—ones that start, stop, set, change, or do something else—then I sit back and try to think about what I'm doing. Ideally, I want all of those actions to happen on the remote computer, not on my local computer.

10.6 But wait, there's more

These examples have all used ad hoc remoting connections, meaning that I specified computer names. If you're going to be reconnecting to the same computer (or computers) several times within a short period of time, you can create reusable, persistent connections to use instead. We'll cover that technique in chapter 18.

I should also acknowledge that not every company is going to allow PowerShell remoting to be enabled—at least, not right away. Companies with extremely restrictive

security policies may, for example, have firewalls on all client and server computers, which would block the remoting connection. If your company is one of those, see if an exception is in place for Remote Desktop Protocol (RDP). I find that's a common exception, because Administrators obviously need some remote connectivity to servers. If RDP is allowed, try to make a case for PowerShell remoting. Remoting connections can be audited (they look like network logins, much like accessing a file share would appear in the audit log), and they're locked down by default to only permit Administrators to connect. It's not that different from RDP in terms of security risks, and it imposes much less overhead on the remote machines than RDP does.

10.7 Common points of confusion

Whenever we start using remoting in a class that I'm teaching, there are some common problems that crop up over the course of the day:

- Remoting only works, by default, with the remote computer's real computer name. You can't use DNS aliases or IP addresses.
- Remoting is designed to be more or less automatically configuring within a domain. If every computer involved, and your user account, all belong to the same domain (or trusting domains), things will work great. If not, you'll need to run `help about_remote_troubleshooting` and dig into the details.
- When you invoke a command, you're asking the remote computer to launch PowerShell, run your command, and then close PowerShell. The next command you invoke on that same remote computer will be starting from scratch—anything that was run in the first invocation will no longer be in effect. If you need to run a whole series of related commands, put them all into the same invocation.
- Make absolutely certain that you're running PowerShell as an Administrator, especially if your computer has User Account Control (UAC) enabled. If the account you're using doesn't have Administrator permissions on the remote computer, then use the `-credential` parameter of `Enter-PSSession` or `Invoke-Command` to specify an alternative account that does have Administrator permissions.
- If you're using a local firewall product other than the Windows Firewall, `Enable-PSRemoting` won't set up the necessary firewall exceptions. You'll need to do so manually. If your remoting connection will need to traverse a regular firewall, such as one implemented on a router or proxy, then it'll also need a manually entered exception for the remoting traffic.
- Don't forget that any settings in a Group Policy object (GPO) override anything you configure locally. I've seen administrators struggle for hours to get remoting working, only to finally discover that a GPO was overriding everything they did. In some cases, that GPO was put into place a long time ago by a well-meaning colleague, who had long since forgotten it was there. Don't assume that there's no GPO affecting you; check and see for sure.

10.8 Lab

It's time to start combining some of what you've learned about remoting with what you've learned in previous chapters. See if you can accomplish these tasks:

1. Make a one-to-one connection with a remote computer. Launch Notepad.exe. What happens?
2. Using `Invoke-Command`, retrieve a list of services that aren't started from one or two remote computers. Format the results as a wide list. (Hint: It's okay to retrieve results and have the formatting occur on your computer—don't include the `Format-` cmdlet in the commands that are invoked remotely).
3. Use `Invoke-Command` to get a list of the top ten processes for virtual memory (VM) usage. Target one or two remote computers, if you can.
4. Create a text file that contains three computer names, with one name per line. It's okay to use the same computer name three times if you only have access to one remote computer. Then use `Invoke-Command` to retrieve the 100 newest Application event log entries from the computer names listed in that file.

10.9 Ideas for on your own

One of the PowerShell modules included in Windows 7 is TroubleshootingPack, which provides command-line access to the new troubleshooting pack functionality in the operating system. I always tell my students and clients to consider enabling PowerShell remoting on all of their client computers, in part because it gives you remote command-line access to those troubleshooting packs. When a user calls for help, rather than walking them through a wizard over the phone, you can just remote in and run the same wizard, in command-line form rather than GUI form, yourself.

If you have access to a remote Windows 7 computer, enable remoting on it. Initiate a one-to-one session and import the TroubleshootingPack module. Then see if you can get and invoke a troubleshooting pack. Remember, run `get-command -module troubleshootingpack` to see a list of cmdlets in that module (there are only two), and run help on those cmdlets to see how they work. You have to provide a file path to the troubleshooting pack you want; you'll find them in \Windows\Diagnostics by default.

Being able to remotely execute these troubleshooting packs—which can even take corrective action if a problem is found—is a strong argument for enabling remoting on client computers, especially those running Windows 7.

WINDOWS ADMINISTRATION

Learn WINDOWS POWERSHELL IN A MONTH OF LUNCHES

DON JONES

Free ebook
SEE INSERT

Windows has so many control panels, consoles, APIs, and wizards it's really hard to keep track of all the locations and settings you'll need. PowerShell is a godsend: it provides a single, unified administrative command line. It accepts and executes commands immediately. And it has in-built language features that will let you write scripts to control any Windows component, including servers like Exchange, IIS, and SharePoint.

Learn Windows PowerShell in a Month of Lunches is a newly designed tutorial for system administrators. Just set aside one hour a day—lunchtime would be perfect—for a month, and you'll be automating administrative tasks in a hurry. Author Don Jones combines his in-the-trenches experience with a unique teaching style to help you master the effective parts of PowerShell quickly and painlessly.

What's Inside

- Learn PowerShell 2—no experience required!
- Concise lessons for busy administrators
- Practical examples and techniques in every lesson

About the Author

Don Jones is a PowerShell MVP, speaker, and trainer. He developed the Microsoft PowerShell courseware and has taught PowerShell to more than 20,000 IT pros. Don writes the PowerShell column for TechNet Magazine and blogs for WindowsITPro.com.

"A seminal guide to PowerShell. Highly recommended."
—Ray Booysen, BNP Paribas

"The book I wish I'd had when I started PowerShell."
—Richard Siddaway, Serco

"Tons of useful exercises allow powerful hands-on learning."
—Chuck Durfee
Graebel Companies

"Whether a beginner or an intermediate, you'll need *no other* book."
—David Moravec
PowerShell.cz

For access to the book's forum and a free ebook for owners of this book, go to manning.com/LearnWindowsPowerShellinaMonthofLunches

ISBN-13: 978-1-617290213
ISBN-10: 1-617290211
54499
9 781617 290213

MANNING US $44.99/CAN $51.99 [INCLUDING EBOOK]