brief contents

**PART 1** PowerShell Administration..........................1

1  ▪ Diagnosing and troubleshooting PowerShell remoting  3
2  ▪ CIM sessions  22
3  ▪ Collecting and analyzing performance counter data  31
4  ▪ TCP port communications with PowerShell  51
5  ▪ Managing systems through a keyhole  71
6  ▪ Using PowerShell to audit user logon events  80
7  ▪ Managing and administering a certification authority database with PowerShell  93
8  ▪ Using PowerShell to reduce Active Directory token bloat  115

**PART 2** PowerShell Scripting ...............................127

9  ▪ The 10 PowerShell scripting commandments  129
10 ▪ Avoiding the pipeline  150
11 ▪ A template for handling and reporting errors  161
12 ▪ Tips and tricks for creating complex or advanced HTML reports with PowerShell  171
13 ▪ Using and “abusing” dynamic parameters  192
14 ▪ PowerShell type formatting  207
15 ▪ Scalable scripting for large data sets: pipeline and database techniques  221
16 ■ Building your own WMI-based cmdlets  236
17 ■ Turning command-line tools into PowerShell tools  247

PART 3  PowerShell for Developers.................................259
18 ■ Using Source Control Software with PowerShell  261
19 ■ Inline .NET code  270
20 ■ PowerShell and XML: better together  278
21 ■ Adding automatic remoting to advanced functions and cmdlets  306
22 ■ Taming software builds (and other complicated processes) with psake  326

PART 4  PowerShell Platforms ........................................341
23 ■ PowerShell and the SQL Server provider  343
24 ■ Creating flexible subscriptions in SSRS  354
25 ■ Inventory database table statistics using PowerShell and SQL Server Management Objects  368
26 ■ WSUS and PowerShell  379
27 ■ Provisioning IIS web servers and sites with PowerShell  392
28 ■ Active Directory Group Management application  406
As terrific as PowerShell is as an automation engine and management tool, and despite the new cmdlets that shipped with Windows 8 and Windows Server 2012, there are still times when a command-line (CLI) tool gets the job done. Perhaps there isn’t a cmdlet replacement yet, or the CLI tool is easy to run. Or perhaps you have some legacy batch files that you need to use. The downside is that although you can run these tools in PowerShell, you’re limited with regard to what you can do with them because CLI tools write text and PowerShell is all about objects.

In this chapter I’ll show you how to turn output from CLI tools into something you can work with in a PowerShell pipeline. Ultimately, I think you’ll want to create your own PowerShell functions that wrap around a legacy command yet still write objects to the pipeline.

**Requirements**

Unfortunately, not every command-line tool lends itself to a PowerShell conversion. The tips and techniques I’ll show you rely on a few key requirements. If the tool you want to use doesn’t fit into the following categories you’ll have a hard time transforming the results to a PowerShell-friendly format:

- *The CLI tool must write text using standard out (StdOut).* One easy way to test this is to run your command and redirect it to a file: `netstat > e:\temp\netstat.txt`. If you can’t do this, the command isn’t a good candidate.
- *The CLI tool should produce formatted or predictable output.* The more structured the output, the easier the transition to PowerShell.
- *The CLI tool should run without any user interaction.* By that I mean you should be able to run the tool with any necessary parameters, press Enter, and get results. If the command requires user interaction such as answering a prompt, it isn’t a good candidate.
You must be able to run the CLI tool in PowerShell. It should go without saying, but the whole point is to run the command in PowerShell and then use the results in the pipeline. If the command you want to run has some sort of odd syntax that makes it difficult to execute from a PowerShell prompt, it will be that much harder to transform.

Most Microsoft command-line utilities that I’m aware of should fall into these categories and meet the requirements. But there might be a third-party command-line tool you use that is a little odd. I’m not saying it’s impossible to “PowerShell-ize” the output; it just might be a little more difficult.

Table 1 shows a number of common Microsoft command-line tools you may want to turn into PowerShell tools.

<table>
<thead>
<tr>
<th>Table 1 Potential PowerShell CLI tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipconfig</td>
</tr>
<tr>
<td>tasklist</td>
</tr>
<tr>
<td>nbtstat</td>
</tr>
<tr>
<td>net [user</td>
</tr>
<tr>
<td>quser</td>
</tr>
</tbody>
</table>

Conversion techniques

Let me state up front that there is no universal technique for transforming the results of a CLI tool into something PowerShell can use. Depending on the tool you may need to combine these techniques:

- Using PowerShell data formats
- Parsing text output

I also think you’ll find it easier to transform a command-line tool into PowerShell using a script or function. Although it’s possible to execute many of the examples in this chapter directly in the console, trust me: a script will be much easier because you’ll rarely accomplish the conversion with a one-line technique.

Looking for PowerShell data formats

The first technique is an easy one. Look at the command help, and see if the output can be formatted into something that PowerShell already knows how to use, such as CSV or XML. For example, if you look at help for driverquery.exe, as shown in figure 1, there are options to format the results to a CSV format.

This is a useful command-line option because PowerShell already knows how to turn CSV data into objects. In fact, with a single PowerShell expression, you can take the CSV output from driverquery and turn it into PowerShell:
Conversion techniques

Figure 1  DriverQuery.exe help with the format option for CSV

PS C:\> driverquery /fo csv | convertfrom-csv

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Display Name</th>
<th>Driver Type</th>
<th>Link Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1394ohci</td>
<td>1394 OHCI Compliant H...</td>
<td>Kernel</td>
<td>11/20/2010 5:44:56 AM</td>
</tr>
<tr>
<td>ACPI</td>
<td>Microsoft ACPI Driver</td>
<td>Kernel</td>
<td>11/20/2010 4:19:16 AM</td>
</tr>
<tr>
<td>AcpiPmi</td>
<td>ACPI Power Meter Driver</td>
<td>Kernel</td>
<td>11/20/2010 4:30:42 AM</td>
</tr>
<tr>
<td>adp94xx</td>
<td>adp94xx</td>
<td>Kernel</td>
<td>12/5/2008 6:54:42 PM</td>
</tr>
<tr>
<td>adpahci</td>
<td>adpahci</td>
<td>Kernel</td>
<td>5/1/2007 1:30:09 PM</td>
</tr>
<tr>
<td>adpu320</td>
<td>adpu320</td>
<td>Kernel</td>
<td>2/27/2007 7:04:15 PM</td>
</tr>
<tr>
<td>AFD</td>
<td>Ancillary Function Dr...</td>
<td>Kernel</td>
<td>12/27/2011 10:59:20 PM</td>
</tr>
</tbody>
</table>

Once the data has been converted you have objects that you can work with, which means you can do whatever you want with the results. Here are some more examples:

PS C:\> driverquery /fo csv | convertfrom-csv | Group 'Driver Type' -NoElement

Count Name
----- ----
279 Kernel
29 File System

PS C:\> driverquery /fo csv | convertfrom-csv | where {$_."Driver Type" -match "File System"} | sort Displayname
Turning command-line tools into PowerShell tools

Chapter 17

Module Name | Display Name | Driver Type | Link Date
--- | --- | --- | ---
Npfs | Npfs | File System | 7/13/2009 7:19:48 PM
Ntfs | Ntfs | File System | 3/10/2011 10:39:3...
rdbss | Redirected Buffering ... | File System | 11/20/2010 4:27:5...
NetBIOS | NetBIOS Interface | File System | 7/13/2009 8:09:26 PM

If all you need is basic PowerShell like I’ve shown here, this should suffice. But there are some potential drawbacks.

First, the property names are taken from the CLI output and might contain spaces. Thus you’ll end up with properties like Module Name and Driver Type. If all you want is pretty output this probably doesn’t matter. But when you want to do something with a property you have to remember to enclose it in quotes, as I had to with "Driver Type".

Second, all properties are treated as strings. Again, depending on the CLI tool, that may not matter. But in my example, if you want to filter or sort on 'Link Date' you won’t get the results you expect—at least, not without a little extra work.

Finally, sometimes the property value includes more than meets the eye. Did you notice in the second example, in which I filtered on the driver type for "File System", that I used -match and not -eq? That’s because the actual value has a trailing space that’s introduced somewhere in the conversion. It isn’t apparent at first glance, until you try to use -eq and wonder why nothing is written to the pipeline. So I used -match, which is more forgiving.

One way to get around the property-name issue is to specify an alternate header in the CSV file:

```
PS C:\> $h="ModuleName","DisplayName","DriverType","LinkDate"
PS C:\> driverquery /fo csv | select -Skip 1 | convertfrom-csv -Header $h
```

I skipped the first line of the CSV output and told ConvertFrom-Csv to use my alternate header. Now my property names are easier to work with.

Handling the property type isn’t too difficult, depending on what you need to do. For one-time formatting you might be able to use a cmdlet:

```
PS C:\> driverquery /fo csv | select -Skip 1 | convertfrom-csv -Header $h | `where {$_._Drivertype -match "file"} | `sort @{Expression={$_.LinkDate -as [datetime]}}
```

Here I took my converted data and sorted on a hash table that returned the LinkDate property as a datetime object. This gets trickier if you want to persist data in this format. Doing so takes a few extra steps, but it can also help with those odd spaces that might crop up in property values. The code in the following listing summarizes everything I just demonstrated.

### Listing 1 Converting CLI CSV output to PowerShell

```powershell
#requires -version 3.0
$hash=[ordered]@{
    ModuleName=[string]
}
```

Create an ordered hash table with property names and types
Conversion techniques

Defined name: [string]
DriverType: [string]
LinkDate: [datetime]

$data=driverquery /fo csv | select -Skip 1 | convertfrom-csv -Header @{$hash.keys}

for ($i=0;$i -lt $data.count;$i++) {
    foreach ($property in @{$hash.keys}) {
        #update each property with a trimmed version
        $data[$i].$property=$data[$i].$property.Trim() -as $hash.$property
    }
}

#write the data to the pipeline
$data

This technique requires you to know in advance what your output will be and what object types you want to define. You build a hash table with the new property names and assign a type as the corresponding value for each key. Fortunately, PowerShell 3.0 offers ordered hash tables, which make this much easier. You use the hash-table keys for your new header, saving all the results to a variable.

In order to clean up $data you have to go through every object and reassign a new value. The new value is the old value trimmed of any trailing or leading spaces and then cast back as the appropriate type.

The end result is that $data now has trimmed and properly typed objects. This means you can use $data in the pipeline like any other cmdlet output:

$data | where DriverType -eq "File System" | sort LinkDate | select -first 5

As you can imagine, this technique is much easier to work with in a script.

Parsing text output

If the CLI tool doesn’t offer a PowerShell-ready format like CSV you have to parse the output. This is where it’s very important that the output is ordered and predictable. There are a number of techniques I think you’ll find yourself turning to:

- Use Select-Object to skip X number of lines. Often a CLI command includes headers you don’t really care about.
- Use Select-String to select command output lines that you want to transform into PowerShell.
- Use the Split() method or operator to break up lines in array. This works well when the command output is structured and predictable.
- Use the Trim() method to clean up values. Leading or trailing spaces can lead to unpredictable results.
- Use regular expressions to either identify the data you want to parse or perhaps identify what property type you want to eventually use. You might use regular
expressions on an entire line of output or on individual array elements if you’ve split up the line.

Let’s demonstrate some of these techniques. I’m going to take the output of `nbtstat /n` and turn it into a PowerShell object. I’ll work through the process interactively so you can follow along; eventually I could take the steps and turn them into an advanced function.

Here’s the original output:

```
PS C:\> nbtstat /n
Local Area Connection:
Node IpAddress: [10.23.36.71] Scope Id: []
NetBIOS Local Name Table
---------------------------------------------
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUARK</td>
<td>&lt;20&gt;</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>QUARK</td>
<td>&lt;00&gt;</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>JDHITSOLUTIONS</td>
<td>&lt;00&gt;</td>
<td>GROUP</td>
</tr>
</tbody>
</table>

Wireless Network Connection:
Node IpAddress: [0.0.0.0] Scope Id: []
```

What matters in this case is the table of registered names. It looks like a good candidate for a PowerShell object. I already see names I can use for property names, like `Name`, `Type`, and `Status`. So first I’ll get the lines of interest:

```
PS C:\> $data=nbtstat /n | Select-String "<"
PS C:\> $data
QUARK <20> UNIQUE Registered
QUARK <00> UNIQUE Registered
JDHITSOLUTIONS <00> GROUP Registered
```

I use `Select-String` to filter and save the results to `$data`. Next, I’ll split each line into an array. The tricky part is that because I used `Select-String`, each line is a `MatchInfo` object, not a string, so I need to use the `Line` property. I’ll also trim off spaces while I’m at it:

```
PS C:\> $lines=$data | foreach { $_.Line.Trim()}
PS C:\> $lines
QUARK <20> UNIQUE Registered
QUARK <00> UNIQUE Registered
JDHITSOLUTIONS <00> GROUP Registered
```

Next I need to split each line on the spaces between words into another array using the `-Split` operator and a regular-expression pattern for a whitespace. Because the order is predictable I know what each array element will be, so I can use them to create a new object:

```
PS C:\> $lines | foreach { $temp=$_ -split "\s+"
  >> New-Object -TypeName PSObject -Property @{
  >> Name=$temp[0]
```
There are potential issues with the techniques I just used. If any property value contained a space I would have had to figure out some other splitting technique. Or if there was a line that had no space between values I would have needed some other splitting technique.

**TIP**  I strongly recommend defining property names without spaces or special characters like commas and apostrophes. You may need to use the Replace() method to strip them out. While you’re at it, my other best practice recommendation is to trim strings. Also note that if your output has values that you might want to sort on or perhaps measure in some way, it’s important that you attempt to cast the text value into the appropriate object type.

### Handling CLI errors

Handling errors from your CLI tool and turning them into PowerShell is something you’ll have to handle on a case-by-case basis. Assuming you’re building a PowerShell script or function to wrap around your CLI tool, you’ll need to include some logic to validate data. For example, let’s say you’re writing a PowerShell function to wrap around the **NET USER** command. When there’s no error you can use the techniques I’ve shown you to parse the output into a PowerShell object. But what if someone tries to get an invalid user?

I would probably do something like this:

```
$data=net user $username 2>$env:temp\err.txt
If ($data) {
    #add your code to process the data
}
Else {
    #add your code to parse the err.txt file to display and handle the error.
}
```

Or you can use a Try/Catch block. Because the CLI tool isn’t a cmdlet it can’t throw a terminating exception. But you can set the error action preference in the Try script-block to handle that for you:

```
Try {
    $ErrorActionPreference="stop"
    $data=net user $username
}
```
A practical example

Let's look at one more practical example. I want to take the output of `ipconfig /displaydns` and turn it into PowerShell output. There's some potentially useful information here that would be easier to work with if I had PowerShell objects.

First, what type of output do I get?

PS C:\> ipconfig /displaydns

Windows IP Configuration

ntp0.cornell.edu
----------------------------------------
Record Name . . . . . : ntp0.cornell.edu
Record Type . . . . . : 5
Time To Live . . . . : 80983
Data Length . . . . . : 8
Section . . . . . . . : Answer
CNAME Record . . . . : dns3.cit.cornell.edu
coredc01.jdhlab.local
----------------------------------------
Record Name . . . . . : COREDC01.jdhlab.local
Record Type . . . . . : 1
Time To Live . . . . : 3583
Data Length . . . . . : 4
Section . . . . . . . : Answer
A (Host) Record . . . : 172.16.10.190
manning.com
----------------------------------------
Record Name . . . . . : manning.com
Record Type . . . . . : 1
Time To Live . . . . : 399
Data Length . . . . . : 4
Section . . . . . . . : Answer
A (Host) Record . . . : 68.180.151.75
manning.com
----------------------------------------
Record Name . . . . . : manning.com
Record Type . . . . . : 2
Time To Live . . . . : 85593
Data Length . . . . . : 8
Section . . . . . . . : Answer
NS Record . . . . . . : yns2.yahoo.com
Record Name . . . . . : manning.com
Record Type . . . . . : 2
A practical example

I could create objects that would include all these properties, but I’ve decided all I really need is the record name and the PTR or A record—essentially, anything that has record in the name. After some initial testing I realize I need to select lines that have the word Record followed by a space:

```powershell
PS C:\> $data=ipconfig /displaydns | select-string "Record 
```

This should give me three lines of data for each record:

```powershell
PS C:\> $data[0..2]
Record Name . . . . . : JDHIT-DC01.jdhitsolutions.local
Record Type . . . . . : 1
A (Host) Record . . . : 172.16.10.1
```

The challenge is to go through $data and group by threes. A For loop will work as long as I increase my counter by three every time instead of the usual one. Assuming I don’t get any odd entries, this command should verify that the every third record is a record name:

```powershell
PS C:\> for ($i=0;$i -lt $data.count;$i+=3) {$data[$i]}
```

```powershell
Record Name . . . . . : JDHIT-DC01.jdhitsolutions.local
Record Name . . . . . : ntp0.cornell.edu
Record Name . . . . . : JDH-NVNAS.jdhitsolutions.local
Record Name . . . . . : COREDC01.jdhlab.local
Record Name . . . . . : maming.com
Record Name . . . . . : maming.com
Record Name . . . . . : maming.com
Record Name . . . . . : yns2.yahoo.com
Record Name . . . . . : yns1.yahoo.com
Record Name . . . . . : incsrc.manningpublications.com
Record Name . . . . . : powershell.com
```

Good. Because I know what the order will be I can build an ordered hash table (which is a new PowerShell v3 feature) for each group of record data and then use those hash tables as properties for New-Object. In v3 I could also use [pscustomobject] instead of New-Object:

```powershell
for ($i=0;$i -lt $data.count;$i+=3) {
    $hash= [ordered]@{
        Name=$data[$i].toString().Split(":")[1].Trim()
        Type=($data[$i+1].toString().Split(":")[1].Trim()) -as [int]
        Value=$data[$i+2].toString().Split(":")[1].Trim()
    }
    New-Object -TypeName PSObject -Property $hash
}
```
Each item is a `MatchInfo` object that I need to convert to a string. Each string is then split into an array on the colon. The property value will be the second item in the array, so I trim it up. I know I want the type to be an `[int]` so I cast it accordingly. Figure 2 depicts the results.

Now that I have rough PowerShell code that works, I can go ahead and build an advanced function to turn this into a PowerShell tool, as shown in the next listing. My function requires PowerShell v3 because I’m using an ordered hash table. If you remove the `[ordered]` attribute it should work in PowerShell v2.

```
#requires -version 3.0
Function Get-IPConfigDNS {
  [cmdletbinding()]
  Param()

  Write-Verbose "Getting DNS cache information"
  $data=ipconfig /displaydns | select-string "Record "

  Write-Verbose ("Retrieved {0} entries" -f $data.count)
  Write-Verbose ("There should be {0} dns records" -f ($data.count/3))

  for ($i=0;$i -lt $data.count;$i+=3) {
    Write-Verbose $data[$i]
  }
}
```

---

**Figure 2** Converting a group of lines into PowerShell objects

---
As I hope you’ve seen, it isn’t necessarily difficult to transform a CLI tool into a PowerShell-based tool. The easiest approach is to use a CLI tool that writes results in a PowerShell-friendly format like CSV. Barring that, look for a tool that writes a predictable and structured format so that you can parse the results into objects. In any event, always look at help for the command you want to convert.
Once you’ve mastered the techniques I’ve demonstrated here you might want to take your PowerShell tools further by incorporating custom formatting or type extensions. Or you could incorporate similar command-line tools into a module. But most important, I hope you’ll share your work with the PowerShell community.

About the author

Jeffery Hicks is a Microsoft MVP in Windows PowerShell, a Microsoft Certified Trainer, and an IT veteran with over 20 years of experience, much of it spent as an IT consultant specializing in Microsoft server technologies. He works today as an independent author, trainer, and consultant. Jeff writes the popular Prof. PowerShell column for MPCMag.com and is a regular contributor to the Petri IT Knowledgebase and 4SysOps.
PowerShell has permanently changed Windows administration. This powerful scripting and automation tool allows you to control virtually every aspect of Windows and most Microsoft servers like IIS and SQL Server. Here’s your chance to learn from the best in the business.

**PowerShell Deep Dives** is a trove of essential techniques and practical guidance. It is rich with insights from experts who won them through years of experience. The book’s 28 chapters, grouped in four parts (Administration, Scripting, Development, and Platforms), were hand-picked by four section editors: Jeffery Hicks, Richard Siddaway, Oisín Grehan, and Aleksandar Nikolić.

**What’s Inside**
- Managing systems through a keyhole
- The Ten Commandments of PowerShell scripting
- Scalable scripting for large datasets
- Adding automatic remoting
- Provisioning web servers and websites automatically to IIS 8
- And 23 more fantastic chapters

Whether you’re just getting started with PowerShell or you already use it daily, you’ll find yourself returning to this book over and over.

**The Authors**
Editors Jeffery Hicks, Richard Siddaway, Oisin Grehan, and Aleksandar Nikolić are joined by PowerShell experts Chris Bellée, Bartek Bielawski, Robert C. Cain, Jim Christopher, Adam Driscoll, Josh Gavant, Jason Helmick, Don Jones, Ashley McGlone, Jonathan Medd, Ben Miller, James O’Neill, Arnaud Petitjean, Vadims Podans, Karl Prosser, Boe Prox, Matthew Reynolds, Mike Robbins, Donabel Santos, Will Steele, Trevor Sullivan, and Jeff Wouters.

Manning Publications and the authors of this book support Save the Children at [www.savethechildren.org](http://www.savethechildren.org).