QUEST SOFTWARE
Smart Systems Management

PowerShell in the Enterprise: Best Practices and Recommendations

Written by
Jeff Hicks
Microsoft MVP and Windows Server Consultant

WHITE PAPER

© 2010 Quest Software, Inc.

ALL RIGHTS RESERVED.

This document contains proprietary information protected by copyright. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording for any purpose without the written permission of Quest Software, Inc. (“Quest”).

The information in this document is provided in connection with Quest products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Quest products. EXCEPT AS SET FORTH IN QUEST'S TERMS AND CONDITIONS AS SPECIFIED IN THE LICENSE AGREEMENT FOR THIS PRODUCT, QUEST ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL QUEST BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF QUEST HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Quest makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Quest does not make any commitment to update the information contained in this document.

If you have any questions regarding your potential use of this material, contact:

Quest Software World Headquarters

LEGAL Dept

5 Polaris Way

Aliso Viejo, CA 92656

www.quest.com

E-mail: **legal@quest.com**

Refer to our Web site for regional and international office information.

Trademarks

Quest, Quest Software, the Quest Software logo, AccessManager, ActiveRoles, Aelita, Akonix, AppAssure, Benchmark Factory, Big Brother, BridgeAccess, BridgeAutoEscalate, BridgeSearch, BridgeTrak, BusinessInsight, ChangeAuditor, ChangeManager, Defender, DeployDirector, Desktop Authority, DirectoryAnalyzer, DirectoryTroubleshooter, DS Analyzer, DS Expert, Foglight, GPOADmin, Help Desk Authority, Imceda, IntelliProfile, InTrust, Invirtus, iToken, I/Watch, JClass, Jint, JProbe, LeccoTech, LiteSpeed, LiveReorg, LogADmin, MessageStats, Monosphere, MultSess, NBSpool, NetBase, NetControl, Npulse, NetPro, PassGo, PerformaSure, Point,Click,Done!, PowerGUI, Quest Central, Quest vToolkit, Quest vWorkSpace, ReportADmin, RestoreADmin, ScriptLogic, Security Lifecycle Map, SelfServiceADmin, SharePlex, Sitraka, SmartAlarm, Spotlight, SQL Navigator, SQL Watch, SQLab, Stat, StealthCollect, Storage Horizon, Tag and Follow, Toad, T.O.A.D., Toad World, vAutomator, vControl, vConverter, vFoglight, vOptimizer, vRanger, Vintela, Virtual DBA, VizionCore, Vizioncore vAutomation Suite, Vizioncore vBackup, Vizioncore vEssentials, Vizioncore vMigrator, Vizioncore vReplicator, WebDefender, Webthority, Xaffire, and XRT are trademarks and registered trademarks of Quest Software, Inc in the United States of America and other countries. Other trademarks and registered trademarks used in this guide are property of their respective owners.

February 2010

Contents

Introduction

With the arrival of Windows PowerShell 2.0, PowerShell is on the brink of widespread enterprise adoption. Microsoft's Common Engineering Criteria dictates that new server and product management tools must be based on Windows PowerShell, so it's not a question of *if* companies will be using PowerShell, only a matter of *when*.

The challenge facing many organizations, though, is to properly manage PowerShell so that it is secure, easy to use, and effective. This white paper discusses the role of PowerShell in the IT environment, describes how to benefit from PowerShell, and suggests best practices for PowerShell scripting.

What is Windows PowerShell?

Let's first examine what PowerShell is, the problem it solves and why it should matter to you. This section also includes a crash course in Windows management and how we got to where we are today.

A Brief History of Windows Management

Until recently, managing Windows was a disjointed affair. There was no single tool to manage everything; you could use resource kit command-line tools for some tasks, and take advantage of scripting solutions using batch files, VBScript, Perl and other languages. You had to continually learn new tools and techniques. You might use a series of command-line tools to manage servers, but one tool might use */computername* and another might use *-machine* as parameters. To help, Microsoft gave us the Microsoft Management Console (MMC), which provided a graphical management interface, and Windows Management Instrumentation (WMI), which could be exploited through scripting. In the end, though, there were still some bits of Windows that were close to unmanageable.

Realizing the challenges administrators faced, especially in larger enterprises, Microsoft responded with a new tool inspired by the Unix management interface, where everything can be managed from a command line. PowerShell v1.0, released in late 2006, enabled administrators to easily manage their Windows environment from a command-line interface using a consistent, easy-to-use, and discoverable tool based on the .NET Framework.

Although many administrators immediately recognized the value of PowerShell, there was some misunderstanding and confusion about its nature and capabilities. In particular, PowerShell is an object-based shell; it is not text based, like Unix . Understanding this difference is critical to fully understanding and using PowerShell.

PowerShell Basics

Windows PowerShell is primarily an interactive management shell: you type a command and something happens. Its core unit of functionality is a *cmdlet* (pronounced "command-let"). This single-purpose command works with *objects*. As with other shells, you can string cmdlets together in a pipeline, passing output from one to the other. Consider the following expression:

```
PS C:\> ps –comp SERVER02| where {$_.workingset –gt 10MB} | SortWorkingSet –desc | Select –First 10
```

Instead of passing text between commands, this expression is passing *process objects* to a series of cmdlets designed to work with objects and their properties. This particular example gets all processes from SERVER02 where the working set size is greater than 10 MB, sorts the resulting objects on the workingset property in descending order, and then selects the first 10 objects in the list. You "see" text at the end, but that is really a textual representation of the objects coming out of the pipeline.

PowerShell's cmdlets follow a consistent and standardized VERB-NOUN naming convention, but an *alias* is often substituted at a command prompt. Cmdlets are single-task commands that are strung together in a pipelined expression. Thus, if you need to sort, you call the **Sort-Object** cmdlet.

PowerShell parameters use the following form: *–parametername value*. Parameter names are also consistent, at least across Microsoft cmdlets. For example, it is always *–computername*; not *–computername* in one cmdlet and *-machine* in another.

As you can see, Windows PowerShell was developed with discoverability and ease of use in mind. Complete cmdlet help with examples is available from the PowerShell commandline by typing *get-help* followed by the cmdlet's name, and cmdlets like **Get-Member** make it easy to discover an object's properties and methods. Windows PowerShell is a surprisingly easy shell to learn on your own.

Scripting and Automation Environment

With PowerShell, Microsoft also wanted to support automation. When PowerShell first came out, many people thought it was merely another scripting language like VBScript. That isn't true: you can manage Windows from a PowerShell command prompt without writing a PowerShell script. But as your management needs become more complex, or you tire of retyping regularly-used commands, you'll want to turn to scripting.

A Windows PowerShell script is simply a text file with a file extension of *.ps1*. Any command you can type at a PowerShell prompt can be pasted into a script. PowerShell's scripting language consists of a few keywords and constructs like If..Else and ForEach. Because PowerShell is manipulating objects, there is no need to learn VBScript-like functions. You can use either a cmdlet or an object's properties directly.

Deploying Windows PowerShell

PowerShell 2.0 was included as an optional feature with Windows Server 2008. Starting with Windows 7 and Windows Server 2008 R2, PowerShell is enabled by default. There are a few ways to deploy Windows PowerShell 2 with the remote management framework. Installation packages for Windows XP and later are available through Windows Update. If you are running a Windows Software Update Services (WSUS) server, you can approve the update and roll it out. You can also follow the download links from Microsoft Knowledge Base article KB968929 (http://support.microsoft.com/kb/968929); there are 32-bit and 64-bit versions, depending on your operating system. You'll need to automate the deployment through logon scripts.

Once PowerShell has been deployed, you should use Group Policy to configure the script execution policy, as described below, as well as any custom WinRM settings appropriate for your domain.

Securing Windows PowerShell

When Microsoft announced PowerShell, many administrators feared it was another vulnerable scripting language like VBScript and were expecting PowerShell versions of the Melissa and “I Love You” viruses. But Microsoft was ahead of the curve and released a product with a very small attack surface. The focus was to prevent scripted or automated attacks. PowerShell won’t monitor anything typed in an interactive console: if someone has access to a server console with administrative credentials and enters a malicious PowerShell expression, PowerShell will execute it. However, if a user gets a malicious PowerShell script as an e-mail attachment, PowerShell will not run it automatically.

Microsoft secures Windows PowerShell scripting by disabling script execution by default via a script execution policy. Even if you have configured the computer to run a script, you must also specify the path to the script, even if it is in the current directory. This helps prevent command hijacking. And PowerShell supports script signing or digital signatures.

Script Execution Policy

Windows PowerShell maintains a per machine setting for the script execution policy. The default is *Restricted*, which means no scripts (.ps1 files) will be executed. You can type all the interactive commands you want, including copying the contents of a script and pasting it into a PowerShell session, but you cannot simply run a PowerShell script.

At the opposite end of the execution spectrum is the *Unrestricted* setting. PowerShell will run any script without question. This setting is not recommended. At a minimum you should consider using the *RemoteSigned* execution policy. With this setting, PowerShell will execute any script you’ve created locally, but any PowerShell script that is detected as coming from the Internet (i.e., downloaded through e-mail or browser) will not execute. If you want to execute a script, you can copy and paste its contents into a new script or digitally sign it. This enables you to study the script and verify its content and functionality before execution.

However the recommended setting is *AllSigned*. When PowerShell encounters this policy, it will refuse to run any PowerShell script without a valid code signing certificate issued by a root certificate authority trusted by the computer. If you have an Active Directory infrastructure, it is not difficult to install the Certificate Services role and issue code signing certificates to your administrator. Because they are issued by your domain, they are automatically trusted by all domain members.

The execution policy can be set per machine using the **Set-ExecutionPolicy** cmdlet. However, it is more expeditious and secure to use Group Policy.

Digital Signatures

When you sign a script using the **Set-AuthenticodeSignature** cmdlet, a digital signature block is appended to the end of the script. This block serves two purposes. First, it identifies the script author and issuing certificate authority. A signed script doesn’t guarantee it’s safe to execute, but if it is malicious, you should be able to track down the author and CA.

The digital signature also includes a hash of the original script. When you execute a signed script, PowerShell compares the script hash with the value stored in the signature. If they do not match, then the file has most likely been altered since it was signed. This is an excellent way of ensuring script integrity. If you carefully manage your code signing certificates, nobody can modify your production scripts without one.

Least Privilege Use

The best way to secure Windows PowerShell is by adhering to the Least Privilege Use concept: don’t give users more rights or permissions than they need. Administrators should use a non-privileged account. When they need to perform an administrative task, they can specify alternate credentials, which are supported by many PowerShell cmdlets, or start a PowerShell session under alternate credentials.

Getting the Most from Windows PowerShell

The following steps will help you get the most value from Windows PowerShell, and many won't cost you a thing.

Training

You'll get more out of PowerShell in the long run if administrators have access to proper training and resources from the start. This will prevent your administrators from spending a lot of time "spinning their wheels" or going down the wrong path. While there isn't technically a *wrong* way to use PowerShell, some approaches are more correct than others.

PowerShell uses a management paradigm that not every administrator grasps immediately. However, once your administrators understand PowerShell fundamentals, the learning curve for PowerShell is much shorter than other languages. Because much of PowerShell is standardized and consistent, once you know how to use the help, work with the pipeline, use operators and a few core cmdlets, learning additional PowerShell cmdlets is easy.

Training options range from multi-day instructor-led classes to self-study DVDs, with many options in between. At the very least, invest in a few books on the subject for the team or department library, and take advantage of free online training opportunities or webinars.

Use Shared Code

As administrators begin developing PowerShell functions and scripts, the emphasis should be on modularity and re-usability. Administrators should strive to never write the same block of code twice. For example, if a function is created to list the names of all users whose password is about to expire, this code should be placed on a shared network resource and administrators' profiles configured to dot source this function.

You should consider creating a script called TeamFunctions.ps1 that resides on the group drive for the IT administrators. This script will be a collection of internally developed functions; it doesn't accomplish anything other than defining the functions. Whenever a new function is approved, add it to the file. You can do the same thing with any shared aliases or common variables.

In every administrator's profile, add this line:

```
. \\file01\it\scripts\teamfunctions.ps1
```

The next time any administrator starts PowerShell, he or she will have access to all the latest team functions every time a new function is added. To get the latest functions, restart PowerShell or reload the TeamFunctions.ps1 script.

Start with the Shell

Don't jump into PowerShell and begin trying to write a 500-line PowerShell computer and user provisioning script. Instead, start with the shell; anything you can run in the shell can be put into a script. First, verify and understand how the PowerShell command runs in the shell. Once you have the expression working, you can then put it into your script.

Also, don't attempt to write a complex, multi-cmdlet expression all at one time. Write the first part of the expression. Verify that it works and produces the objects you are expecting. Then pipe it to the next part of your planned pipeline. If everything is running as expected, repeat the process until your expression is complete. This may seem to take more time, but in the long run it doesn't. If you have a complex pipelined expression that results in an error, or, even worse, doesn't return the results you are expecting,

you'll spend a lot of time figuring out where things went wrong. But if you start from the beginning with known good expressions, you won't face this problem.

Also be sure to use non-scripting tools. For example, **Get-WMIObject** will run a WMI query interactively from the shell. Use a tool like WBEMTest.exe to test and validate your query, especially if you are connecting to a remote machine. If the query works, you can paste it into PowerShell. If you have problems, you can then concentrate on your PowerShell code because you've already verified that the WMI query is valid. Again, this may seem time-consuming, but for anything but the simplest of queries, your overall development time will be shorter.

Don't Re-Invent the Wheel

Earlier we saw how to share code among the administrators on your team. But you can also think more broadly. PowerShell has been around for several years now, so a growing number of PowerShell script samples are freely available. While you can certainly use your favorite search engine to find code that accomplishes a given task, another good option is PoshCode.com (http://www.poshcode.com), a community library of user submitted PowerShell scripts and functions. While you may not find a script that accomplishes exactly what you need done, you might find something close or that can be a starting point.

There are also a number of PowerShell blogs and other resources that offer PowerShell based scripts and tools, such as my Mr. Roboto column in REDMOND (http://redmondmag.com/articles/list/mr-roboto.aspx). You don't have to start from scratch and develop in a vacuum.

When developing PowerShell scripts, remember to think "modular and re-usable." And, most importantly, don't spend time writing a function if someone on your team already has. Call their function in your script and move on. PowerShell makes it very easy to incorporate external code sources without a lot of cutting and pasting.

Use a Scripting Editor

Finally, make sure you invest in a scripting editor. While PowerShell files are simple text files, Notepad suffices only for the simplest of scripts. You need a script editor with syntax highlighting, command completion, and an integrated debugger. Some editors include a library of PowerShell code snippets you can quickly drop into your script. There are many free and commercial tools on the market, many of which offer a free trial period. Download and try as many as you can, until you find one that offers the features you need at a price point you can afford. An effective script editor will speed up script development and cut down on bugs.

At the very least, use the simple script editor in the PowerShell ISE (Integrated Scripting Environment). According to Jeffrey Snover of the Microsoft PowerShell team, one of the reasons this editor was developed was to provide a Notepad alternative for PowerShell users.

PowerShell Best Practices

If you are going to deploy Windows PowerShell across your enterprise, you should review the best practices described below. These are especially recommended if several administrators will be using PowerShell scripts to manage production assets. Adhering to the best practices will ensure you get the most from your PowerShell investment in a secure and efficient manner.

Digitally Sign All Scripts

Any script that will be executed in a production environment, especially on mission-critical servers, should be signed with a code signing certificate trusted by your domain. You can certainly acquire a code signing certificate from a third-party vendor, but it doesn't take much to set up the free Certificate Services from Microsoft and issue your own. Since the certificates are essentially issued by your domain, they are automatically trusted by all member servers and desktops.

A digitally signed script is critical because it ensures that the script has not been modified since it was last signed. You can use the **Get-AuthenticodeSignature** cmdlet to verify signature integrity.

Use the AllSigned Execution Policy

Use Group Policy to enforce the *AllSigned* execution policy across your domain, or at the very least, across your mission-critical servers. This will prevent any script not signed by a trusted root from executing in your environment. It has no effect on interactive sessions; PowerShell will execute whatever commands are entered interactively that the user has permissions and rights to execute.

However, remember that the script execution policy is just one security element you need to consider. It is not a security boundary, but will help protect you from automated or unattended code execution.

Create and Sign all Profile Scripts

Windows PowerShell looks to four different profile scripts to configure PowerShell:

Profile	Location
All Users for All Hosts	C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
All Users for the Current Host	C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
Current User for All Hosts	C:\Users\XXX\Documents\WindowsPowerShell\profile.ps1
Current User for the Current Host	C:\Users\XXX\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

None of these profiles exists by default. However, if such a file is found, PowerShell will execute any commands as the current user. Even if you aren't using all of these profiles, you should create all of these scripts and digitally sign them (you can insert a comment indicating the script is intentionally empty). That way, if someone or something other than the user modifies any of these profiles, PowerShell will throw an exception when it is executed because the digital signature will be broken. Of course, if someone or something has the ability to create or modify these files without authorization, you have more urgent security needs. But at least they can't use PowerShell to execute malicious commands.

The PowerShell Integrated Scripting Environment (ISA), which is sometimes referred to as graphical PowerShell, has its own set of security profiles:

Profile	Location
All Users for the Current Host	C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1
Current User for the Current Host	C:\Users\Jeff\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1

Implement Scripting Standards

Even if only a single administrator is creating PowerShell scripts, you should develop a standardized script template. The most important part of the template is a commented header. Using a standardized header makes it easier to keep your scripts organized. At an absolute minimum, your template should include a script name, author, date, and version number. Also recommended are a syntax example, usage notes, key words or tags, and a change log. Many script editors support templates so this doesn't have to be a difficult proposition.

In addition to establishing standard file layout, you should use the standard verb-noun naming convention for your scripts and functions. Use a standard verb from the **Get-Verb** cmdlet. The noun should be descriptive, and, if necessary, include a corporate prefix to distinguish it from a similar Microsoft cmdlet. The same is true of parameter names. If there is a parameter used by a Microsoft cmdlet that makes sense for your project, use it. Don't use *–SYS* when *–computername* will work just fine. If you use a unique parameter name, make sure it is long enough to be meaningful. You can always add a parameter alias.

Finally, when you develop PowerShell scripts, use full cmdlet names and parameters. PowerShell has a number of shortcuts that make it easy to run commands from a PowerShell prompt. And while the same command will run in a PowerShell script, there's no penalty for using complete names. You only have to type it once. Consider the following command:

```
ps –c server01 | ? {$_.ws -gt 5MB} | sort ws -Des | select -fi 5 |ft -au | of report.txt
```

This command can be run from a PowerShell prompt, but it will fail if you try to run it in a script because 'of' is an alias defined on a local computer. Unless you also define that alias in your script, the command will fail. The better solution is to use full cmdlet names and parameters, as follows:

```
Get-Process –computername server01 | where-object {$_.workingset -gt 5mb} | sort -Property
workingset -Descending | select-object -First 5 | Format-Table -AutoSize | Out-File report.txt
```

This command will execute the same as the first one, but it is easier to understand and troubleshoot. It may seem like a lot to type, but you only have to do it once, and a good scripting editor will offer command completion; it really isn't much work at all.

Test in a Non-Production Environment

It should go without saying: any PowerShell script or function—whether developed internally or downloaded from a source like PoshCode.org—must be tested thoroughly in a non-production environment.

Be sure to test not only how the script runs successfully, but also how it fails. What happens when you pass it invalid parameters or if a required resource is unavailable? What can you do to make the script or function fail? It is critical that you understand how the script handles problems. This is especially true of internally developed scripts. You should be able to take this information and revise your script to make it as robust as possible.

Using traditional application development methodologies with script development will help make your code better and your testing easier. You should have peer code reviews, unit testing, pilot testing, as well as management sign-off and approval. Administrative scripting does not have to be ad hoc or throwaway; in fact, it's just the opposite, scripting should be closely managed. PowerShell can bring a server or network down with only a few lines of code, assuming proper permissions. You should not have PowerShell scripts running in your environment that you don't understand, trust and approve.

Use Version and Access Control

Another accepted application development concept that you should apply to your PowerShell script development is version control. PowerShell scripts should be checked into a version control system so that you can keep track of changes and always know you are running the most current approved version of a script. After script development and testing, the script should be signed and checked into your source control system. Some scripting editors make this easy by hooking into source control systems.

On a related note, file level access controls for your scripts must be tightly maintained. You must ensure that only appropriate users have access to modify a script file. This is particularly critical if you aren't using a source control mechanism.

Include Script Documentation

As you develop PowerShell scripts, include script documentation from the very beginning. It doesn't have to be complex or long, but should be informative enough for someone else to look at your script and understand what it is doing. All you need are short comments throughout the script explaining what the script is about to do next. For example:

```
#get processes with a workset over 10MB

$procs=get-process | where-object {$_.workingset –gt 10MB}
```

There's no performance penalty and extensive documentation is helpful when debugging or troubleshooting.

On a related note, when creating PowerShell 2.0 scripts or functions, include comment-based help and examples. Treat your script or function like a cmdlet.

Add Debug or Verbose Information to Your Scripts

PowerShell 2.0 lets you create scripts and functions that can behave like a compiled cmdlet. In addition to internal script documentation, add PowerShell commands throughout your script that can be used to display what the script is doing and track variable values. You can insert **Write-Debug** or **Write-Verbose** commands into your script from the very beginning.

```
Write-verbose "Getting processes with a workingset over 10MB"

$procs=get-process | where-object {$_.workingset –gt 10MB}

Write-verbose "Found $($procs.count) processes"
```

These lines will execute only if –verbose is specified; the messages will be written to the verbose pipeline, which makes it easier to track what the script is doing. Again, there's no performance penalty and you can get useful troubleshooting information by simply specifying a parameter. You'll also realize that these commands can double as documentation!

Think "Object"-ively

A major recommendation is to think "objects" when writing PowerShell scripts. Unlike other shell languages, PowerShell does not require you to parse text to get the results you want. When working with services, you aren't manipulating a string representation of a server. You are working with an object that

is tied to the actual service. PowerShell uses the properties and methods of an object, and so you should as well.

A valuable technique is to modify and create your own custom objects in PowerShell. While you can certainly create a PowerShell function to return a simple value, ideally you want to be working with collections of objects that can be written to the pipeline. Whenever possible, have your scripts write objects to the pipeline and where appropriate, accept pipelined input. PowerShell 2.0 makes this very easy to accomplish. For example, suppose you need a script to return quota usage information for a given server. In other scripting languages, such as VBScript, you would have to display several pieces of information (username, quota limit, space used, space free) and then parse all that text into some type of meaningful report. But with PowerShell, you can create a custom object with all of the necessary information and write it to the pipeline. Once it is in the pipeline, you can use other PowerShell cmdlets and tools. The following script would give you quota usage information for your specified server:

```
PS C:\> Get-diskquota jdhit01 -Volume e: | Where {$_.UsedSpaceMB -gt 10} | sort
PercentUsed,Username –desc | export-csv f:\reports\quotareport.csv
```

.

Start Small

Finally, as discussed earlier, don't take on too much work for your first scripts. A first-year medical student shouldn't be expected to perform heart-lung transplant, and a beginning PowerShell user shouldn't try to write the Great American PowerShell script. Start small and master the basic techniques. Over time—and it won't be long—you'll find yourself able to develop more complex scripts that run correctly the first time. You'll also begin building a library of code snippets and functions that you can re-use, again, reducing development time. So while PowerShell has the ability to accomplish some very large, enterprise-level tasks, start small and work your way up.

Conclusion

Windows PowerShell 2.0 is on the brink of widespread enterprise adoption. By following the suggestions and best practices in this white paper, you can enjoy the benefits of PowerShell quickly and securely. A wealth of resources are available, and the PowerShell community and ecosystem is vibrant and growing. Many people are eager to help you get to your PowerShell destination, and we welcome you aboard.

About the Author

Jeffery Hicks (MCSE,MCSA,MCT) is a Microsoft MVP and an IT veteran with almost 20 years of experience, much of it spent as an IT consultant specializing in Windows server technologies. He works today as an independent author, trainer and consultant. Jeff is a columnist for *Redmond Magazine* and *MCPMag.com*. He has co-authored or authored several books, courseware, and training videos on administrative scripting and automation. His latest book is Windows PowerShell 2.0: TFM (SAPIEN Press 2009). Jeff is a moderator at ScriptingAnswers.com, a subject matter expert at TheExpertsCommunity.com and a frequent contributor to many scripting-related forums.
You can follow Jeff at jdhitsolutions.com/blog and twitter.com/jeffhicks.

About Quest Software, Inc.

Now more than ever, organizations need to work smart and improve efficiency. Quest Software creates and supports smart systems management products—helping our customers solve everyday IT challenges faster and easier. Visit **www.quest.com** for more information.

Contacting Quest Software

PHONE 800.306.9329 (United States and Canada)
If you are located outside North America, you can find your local office information on our Web site.

E-MAIL **sales@quest.com**

MAIL Quest Software, Inc.
World Headquarters
5 Polaris Way
Aliso Viejo, CA 92656
USA

WEB SITE **www.quest.com**

Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract.

Quest Support provides around-the-clock coverage with SupportLink, our Web self-service. Visit SupportLink at **https://support.quest.com**.

SupportLink gives users of Quest Software products the ability to:

- Search Quest's online Knowledgebase
- Download the latest releases, documentation, and patches for Quest products
- Log support cases
- Manage existing support cases

View the Global Support Guide for a detailed explanation of support programs, online services, contact information, and policies and procedures.

QUEST SOFTWARE
Smart Systems Management

5 Polaris Way, Aliso Viejo, CA 92656 | PHONE 800.306.9329 | WEB www.quest.com | E-MAIL sales@quest.com
If you are located outside North America, you can find your local office information on our Web site

© 2010 Quest Software, Inc.
ALL RIGHTS RESERVED.

Quest Software is a registered trademark of Quest Software, Inc. in the U.S.A. and/or other countries. All other trademarks and registered trademarks are property of their respective owners.
WPW-PowerShellBestPractices-Hicks-US-MJ20100222