Global Knowledge®

Expert Reference Series of White Papers

Getting the Most Out of Windows PowerShell

Getting the Most Out of Windows PowerShell

Mike Hammond, MCT, MCITP:SA, MCITP:EA, MCSE, MCAD, MCDBA

Introduction

Windows PowerShell represents a dramatic change in the way Microsoft thinks about administering Windows and other products—and indeed administering an organization's entire IT infrastructure. Conversational use of PowerShell has rapidly become a must-have skill for the Windows Server administrator, while mastery of PowerShell is becoming a key to advancement for IT professionals. As a recent PowerShell student of mine put it, "The team's scripting guy is the one person we can't live without."

Microsoft has layered PowerShell functionality under the surface of many of its most well-used graphical interfaces. (See Fig. 1, below) Every task you initiate in the Microsoft Exchange GUI is translated into PowerShell. The same is true for System Center Virtual Machine Manager. If you've used the Active Directory Administrative Center (ADAC) interface to reset a user's password, you've again been using PowerShell without even realizing it. ADAC in Windows Server 2012 and later now has a PowerShell History area at the bottom of the GUI to actually show you the PowerShell statements that the tool executed on your behalf.

As the pace of Microsoft's development efforts increases, their engineers are increasingly developing innovative improvements to their products faster than Microsoft's own GUI development teams can build GUI interfaces for those features. These new features are, in many cases, being made available first through PowerShell, and then GUI functionality is being added in later releases, if at all! Being savvy with PowerShell provides technology advantages that just aren't available to those who can only administer through pointing and clicking.

While many organizations have begun investing in PowerShell skill development in their IT professionals, one important question you should consider is this: am I getting the *most* I can out of PowerShell? Are there capabilities of the shell that have gone unexploited by my organization? Efficiencies that have not yet been taken advantage of? The purpose of this white paper is to survey the areas of untapped potential that many organizations are failing to capitalize upon, and point the way to maximizing the capability of this trail-blazing automation technology.

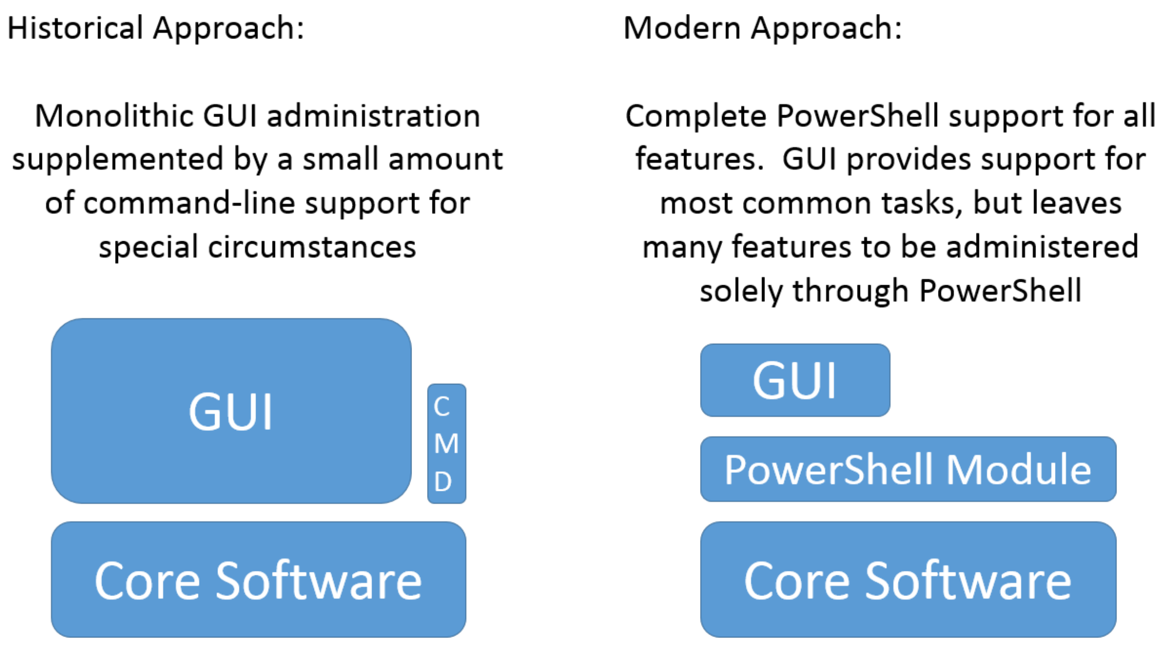
Historical Approach:
Monolithic GUI administration supplemented by a small amount of command-line support for special circumstances
GUI
CMD
Core Software
Modern Approach:
Complete PowerShell support for all features. GUI provides support for most common tasks, but leaves many features to be administered solely through PowerShell
GUI
PowerShell Module
Core Software

Figure 1

Why PowerShell?

So what makes PowerShell so valuable? PowerShell made a quiet entrance in 2006, as a downloadable add-on feature for Windows Server 2003 and clients running XP or Vista. It went largely unnoticed except by existing scripting/automation enthusiasts until its inclusion as a built-in feature of Windows Server 2008 and Windows 7. So what is PowerShell? PowerShell is, at its root, an engine for processing commands—all sorts of commands. The PowerShell engine has been embedded in a command-line interface, bringing extraordinary richness to administrators already familiar with older command prompts like CMD.EXE. It has been embedded in a number of script editor tools, and inside the Windows OS itself to support remote administration through the WinRM service. It lives under the hood of an increasing number of server administration GUIs used by Microsoft products, but also underneath unaffiliated companies' offerings (like NetApp's 'Data ONTAP') and even in competitors' products (like VMWare's 'PowerCLI' and Cisco 'UCS').

What makes PowerShell such a remarkable enabler of IT technologies is the way that it reduces all information and systems that it can manage to a single common denominator: objects. PowerShell thinks in the same object-oriented way that high-end developers do, but it brings that technology effortlessly to the command prompt for day-to-day operations. Extracting information from a computer system produces a standard package of information that can be utilized the same way whether the data is coming from a VMWare server or a Windows 8.1 client.

Objects are easy-to-manipulate packages of information that contain properties that describe that object, and methods that allow actions to be taken upon them. These objects are based on the decade-long success of the .Net Framework, which underpins everything PowerShell does by defining the kinds of objects that it can manipulate. Objects can be easily handed off from one command (known as a cmdlet in PowerShell) to another without loss of fidelity in the information in that object. As such, information about wildly different IT systems can be operated on using the same basic command structure, and insights gained from data extracted from one system can be immediately turned into configuration settings for another, related system.

Perhaps a customer reports sluggish behavior in a system we administer. An admin running a PowerShell command might identify a particular server that is experiencing unusually high processor usage. Another command might be used to identify the presence of a malware executable on that computer. Further commands could take that information as input, terminate that program, and create an AppLocker rule on the whole server farm to prevent that same program from launching on any of the other application servers. In another scenario, a command might extract a list of Active Directory users who haven't changed their passwords in the last 90 days, and pass that information to a command that disables those user accounts.

Getting More

Ask a dozen people what PowerShell is, and some interesting fraction will tell you it's a new, more powerful version of `CMD.EXE`. As we've identified, that's *one* way it can be experienced, out of many. People who are looking at PowerShell this way are viewing it as a toolbox—a container for commands that might be useful from time to time, but can otherwise sit in the toolbox doing nothing. What's missed by those who think of PowerShell as merely a toolbox is its greater significance: as a tool *factory.*

PowerShell provides the infrastructure to allow a moderately-skilled individual to upgrade from merely using combinations of others' commands, to creating his or her own. These user-designed and built tools can support the full range of functionality present in the cmdlets provided by Microsoft or other software developers, including: modular development, rich error handling, nearly effortless documentation, plus support for parameter validation, pipeline input/output, custom formatting, and more.

With these capabilities, an IT team's "scripting guy" (or gal) can respond to the complaint that "there's no easy way to . . . <you fill in the blank>" with a custom-designed PowerShell command that specifically addresses the need of that particular scenario for that specific IT culture in that specific organization.

Modules

Module functionality makes PowerShell extensible. Microsoft and other third-party developers use Visual Studio or other sophisticated programming tools to create modules that support expensive pieces of retail software. Now, that same extensibility is put in the hands of your IT staff, without any of the complexity or expense.

PowerShell is designed to optimize the flexibility of the tools that a user creates. Tools should be easy to create, easy to deploy, and easy to update when revisions are needed. PowerShell accomplishes these goals through the use of script-based modules. A script module, in its simplest configuration, is a folder containing a script file, and is installed in a specially designated Modules folder on a computer through a simple file copy operation. Any single module can contain one or many custom-created commands, with each command designed to automate a specific real-world task in your IT environment.

In PowerShell 3.0 and later (included in WS 2012 and Win 8 or later), PowerShell automatically detects a properly configured module and can instantly make use of the commands found inside it. Those custom-built commands now become just another part of PowerShell, as far as an end user is concerned. Those commands are discoverable in the same way as built-in cmdlets, and can expose the exact same kinds of behavior that built-in cmdlets do.

This process can become iterative. Once an IT staffer has become accustomed to using a custom command built by a coworker, he or she might build a newer, more sophisticated tool that incorporates the use of the older custom command into its operations. As this process repeats, an IT organization's staff can, over time, develop a custom toolset designed with solving (and indeed, preempting) their own most common problems in their own, unique, infrastructure. Suddenly, the organization's most tedious, time-consuming tasks are replaced by automated, even scheduled, tools—freeing up a staffer's valuable time to work on projects that really need the resourcefulness of a dedicated knowledge worker.

Pipelining

One of the centerpieces of PowerShell administration is pipelining. Adapted from an original UNIX idea, pipelining had a small role in batch scripting and actions taken at the classic `CMD.EXE` command prompt. You may recall typing something like "`type ReadMe.txt | more`" to see the contents of a file one page at a time. In PowerShell, that feature takes center stage. Everything PowerShell does is implemented through a pipeline, even tasks that may seem not to need one. In the more obvious cases, a PowerShell command is typed, followed by the pipe character "`|`" and then a second command. The first command produces objects as output, and the pipe carries them to the second command. The second command treats the results of the first command as input. This process can be repeated to set up a chain reaction in three, four, or more commands!

This capability is extended to custom-built commands stored inside modules, as well. A suitably trained individual can configure their own custom command to accept inputs of any appropriate object type from the pipeline, act on those objects, and then emit new objects that describe the results of those actions back onto the pipeline. In some cases, a custom command might utilize the entire incoming object as a single input, or might rely on a different PowerShell technique for extracting individual properties from the incoming object, utilizing a combination of those specific properties that serves the goal of that particular custom command.

A custom tool is capable of creating entirely new objects with a completely user-defined list of properties. Some of these properties can, in fact, be references to collections of other, subsidiary objects, each with their own set of properties! These objects carry a generic "PowerShell Object" data type, but that can be easily replaced with a unique custom object type name as well.

PowerShell enables these mechanisms with a bare minimum of coding effort—frequently, just a few lines of instructions.

Error Handling

Unexpected circumstances are encountered by all software, but well-written software can react to those surprises in ways that preserve the integrity of data, and communicate clearly with the user about what happened and what should be done next. PowerShell implements the very common error handling mechanism available in the .Net Framework, allowing authors to describe the main intended functionality, but also to define their own reactions to problems should they occur. Error message output can be produced automatically, and further verbose output can be made available to a troubleshooter who adds a simple "`-verbose`" parameter to the custom command.

Default error handling behavior can be easily configured inside a custom-developed tool. The creator can quickly specify whether a tool should merely provide an error message and continue functioning in response to a problem, or halt operations and wait for administrator action before resuming. Where appropriate, error conditions can be suppressed, avoiding needless notifications in areas where they aren't beneficial. Error information can be easily shunted to a log file for later analysis, or displayed immediately on the screen—or both!

Many errors can be detected, of course, before a tool goes into production, thanks to a built-in debugger that allows a user to step through the tool's instructions one step at a time, inspecting for problems in the execution of his or her code.

Documentation

As a former C developer, I can attest that there's nothing more anticlimactic after successfully solving a tricky technical problem with code than needing to document the solution. "What?" says the developer, "I just saved the company with this piece of code, and now you want me doing . . . paperwork? Why are you punishing me?" Of course, the instinct to document is the right one—it's essential for the developer (or the one who takes his position one day) to be able to retrace the thought process that led to that piece of code.

A programmer has an array of excuses to rebut this assertion: it's time-consuming; I'm not good at formatting pretty documents; I might make a mistake in the documentation; and no one will ever know. PowerShell demolishes all of these excuses and more. There's now no good reason to avoid documenting our custom PowerShell tools, because PowerShell allows for creation of rich, meaningful documentation with a format that is indistinguishable from Microsoft's own documentation on their own built-in cmdlets, and enables this with the absolute bare minimum of effort on the part of the author.

PowerShell utilizes a "comment-based help" system that doesn't require the developer to manually build a separate document to contain their help file. The same help retrieval command used to retrieve the "help file" for Microsoft's cmdlets is also able to operate on custom tools built by your IT staff. The help system inventories the tool's comments and key portions of the code, dynamically generating a help file with the exact same formatting used by Microsoft's own help content for their own cmdlets.

The basic help file contains a user-authored description paragraph followed by a dynamically-generated syntax diagram based on the technical structure of the tool that the user built. Parameters are automatically flagged as optional or mandatory appropriately. Their data types are automatically described based upon the content of the parameter block inside the code. Parameters are grouped into parameter sets automatically, all based on the live code inside the module. If the code changes, the next retrieval of the help content will contain an updated description of the current version of the code. The technical portions of the help file will necessarily be error-free, because the author doesn't even need to write it! There is no "help file," truly—it's all dynamically generated as needed.

PowerShell's help system allows for two deeper levels of help to be extracted from commands: a "detailed" and a "full" version of the help. The detailed version adds a set of user-authored (but PowerShell-formatted) examples, plus a paragraph of description of each parameter and what it is designed to do in the tool. The full help adds further technical detail about each parameter, identifying default values, use of wildcards, pipeline interaction, and more.

Finally, a PowerShell user can build documentation for its original purpose: to meaningfully inform the next person who comes along about how exactly to use the tool he or she built. To the question, "How do I use your tool?" your IT guy can confidently say, "Just read the help file on it." "I don't know where the help files are stored," complains the new user. "Just run Get-Help on it," explains the IT guy. "What, you mean like the way I do with Microsoft's own cmdlets?" asks the user.

Exactly.

Parameter Validation

A chapter in the middle of Microsoft Press's book, *Writing Secure Code, 2nd Edition* is titled "All Input is Evil!" The point is sharply made to spotlight an important consideration—any data supplied by a user to a program has the potential of being accidentally harmful to that program. Beyond that, some data supplied by users to programs will be part of malicious attempts to break that program in ways that benefit the attacker at the expense of the program's owner. Good developers must assume that any data supplied by a user is untrustworthy and potentially malicious until proven otherwise. Great developers build in safety checks in their tools that prove any incoming data is good before relying on it to make changes to the system.

Once again, developers historically look for excuses for writing validation code. "The data has already been validated by the time it reaches my tool." "The validation introduces performance problems in the code." "I'm not good at writing error messages for all the things that can go wrong." And once again, PowerShell has the answer. It's a trivial change to a custom tool to add validation to its parameters—a simple matter of telling the parameter which type of validation should be done—and PowerShell does the rest. If a `-Priority` parameter should take values of only "`Low`", "`Medium`" and "`High,`" then it's easy to tell the parameter to enforce that, and PowerShell *itself* produces the error message if the validation criteria isn't met. Ditto with criteria including input within a specific numeric range, validating that a text input is of a certain number of characters, validating non-null input, matching regular expression patterns, and other, more sophisticated kinds of validation. The developer need not design validation routines nor retry logic nor politely descriptive error messages.

Building good code has never been easier.

Formatting Output

Another enduring challenge of software development is dealing with making the data produced by the tool usable by non-technical users. Cryptic formatting commands and the demands of developing working data layout algorithms are the bane of first-year computer science students learning C or Java. But not for PowerShell users. Once again, PowerShell's reliance on Objects as the lingua franca of IT administration becomes vitally important. Because all data is contained in objects, and all objects have a known structure, PowerShell already has powerful formatting routines designed to react to any kind of incoming object and produce pleasant, readable output based upon that object.

Depending on the nature of the data being displayed, PowerShell has the flexibility to choose from two main formatting options. The first creates long, linear lists of information—each Property value gets its own line in the output, preceded by the label that names that piece of data. That line is followed by the next property, and the next, until eventually a blank line identifies the end of the description of one object and the beginning of the next. The second main display format creates elegant tables of data—property labels are arranged at the top of the display, with multiple columns of data arranged below. These table layouts are often informed by XML documents called views, which identify exactly which seven or eight properties (out of dozens) should be displayed in the table.

One of the ironies of this effortless built-in formatting behavior is that it makes my job as a PowerShell trainer a little harder! A recurring challenge in my classroom is helping users understand that there is frequently *much* more data available than the seven or eight properties on display following a simple data retrieval command. The data is so elegantly formatted that students assume there's nothing more to see!

The formatting environment provides the user with complete control over the number of properties to display, whether to display the objects in a list or table, how many characters of room are allocated to each column of data, and whether to arrange a column of data left-justified, right-justified, or centered. It even provides support for creating custom properties on-the-fly, calculating a custom value by performing user-specified mathematical or string-manipulation operations on the available properties.

This formatting can be done ad-hoc, or can become a more permanent part of the use of a custom PowerShell module. The same "view" XML file format used internally by PowerShell is also available for more advanced PowerShell toolmakers willing to take on the challenge of mastering the XML description of the table to be created in response to a particular kind of object coming down the pipeline. These view files can be added to a Module to ensure that every time a particular module is used, the objects produced by its tools will be formatted (at least by default) in a pleasant and immediately useful way.

Coming Attractions

The PowerShell world continues to grow rapidly. Recent releases have added yet more advanced functionality.

PowerShell Workflow brings the reliability of Windows Workflow Foundation (formerly available only through Visual Studio) and makes it available in PowerShell. WWF allows for definition of "Activities" that take place on specific servers or on multiple servers, performed in parallel or sequentially. These activities are atomic units of administration—if a server reboots unexpectedly in the middle of performing an activity, WWF can detect that, roll back the uncompleted task, and then start again. This creates an extremely reliable infrastructure for rolling out sophisticated infrastructure changes on an automated basis.

PowerShell Desired State Configuration (DSC) brings declarative programming to the PowerShell world. DSC allows the developer to build a configuration file that describes the way a computer (a server, perhaps) should be configured, and feeds it to the DSC engine. DSC analyzes the server, and where the server doesn't match up with the configuration file, DSC automatically calls the needed PowerShell statements to configure the system in the way that the configuration file specifies.

Constrained Endpoints support delegated administration in PowerShell. A server can be configured to allow a specific administrator to use only a limited subset of normal PowerShell features when connected to that server. Such a system might limit a user to just some of the modules on the system, but not others—or certain commands within a module, but not the rest.

Scheduled Jobs functionality allows any PowerShell command (or script) to be scheduled to be executed at a later time. That might be an execution just once tonight at 11:30 p.m., or every third Tuesday at 3 a.m. with a randomized offset to avoid having multiple servers get hit with a heavy processor load at the exact same time.

Nano Server support: Windows Server 2016 will ship with a new server installation mode called Nano Server. This feature will allow servers to be installed with no graphical interface at all. Yes—even less than the "minimalist" Server Core mode. One interesting thing this server will have, though, is "Core PowerShell"—an environment for providing all the great PowerShell manageability that we've been enjoying for the last near-decade.

Conclusion

There is no aspect of the way Microsoft makes software today that is not impacted by the arrival of PowerShell. Consider that 50,000 virtual network changes are made daily in the Microsoft Azure cloud, according to Microsoft's Products and Enterprise Division manager Jeff Woolsey—and none of those changes are being made manually—they're all software-driven. They have to be—it would be economically ruinous and hugely error-prone to do them manually.

This impact is rippling outward, affecting Microsoft's customers, allies, and competitors alike. As the pace of IT change grows, lessons learned need to be distilled into automated tools that detect your most commonly experienced problems and react to them with your preferred response, integrating the behavior of your server environment, your storage platform, your database resources, your cloud infrastructure, and much more. Your organization simply cannot wait for some software developer to release a perfect management solution for your company—no other company is quite like yours, and no other company has quite your set of challenges or your particular set of opportunities.

But you have a priceless resource at your disposal—a hard-working IT team that knows your system inside and out. They want to make your infrastructure hum like a well-oiled machine, but they only know how to manage it with the graphical tools they've been given. GUI administration has served us well in the past, but in time point-and-click IT operations will struggle to keep up with faster, more agile competitors with pre-scripted solutions to their biggest challenges. Corporate management will be eyeing the market and wondering just why the competition seems to be getting more IT horsepower from a smaller IT budget.

There is one perfect tool for capturing the best practices that your IT staff already implements manually, and transforming that with scripting. That tool provides the ability to cross-reference information from a variety of different sources, and apply custom solutions that marshal your unique resources to solve the problems you face. The tool is easy to learn, included for free with all your Windows OSs, and provides enormously rich possibilities to grow skills over time.

Have you unlocked the gates to the tool factory in your IT organization? Is it time for you to Get-PowerShell?

Learn More

Learn more about how you can improve productivity, enhance efficiency, and sharpen your competitive edge through training.

Automating Administration with Windows PowerShell (M10961)

Windows PowerShell Scripting and Toolmaking (M55039)

PowerShell Essentials

Visit **www.globalknowledge.com** or call **1-800-COURSES (1-800-268-7737)** to speak with a Global Knowledge training advisor.

About the Author

Mike Hammond is a Microsoft Certified Trainer who's delivered technical training for over 14 years, focusing on infrastructure management and scripting technologies. Mike has a bachelor's degree in IT and Software Engineering and lives in the Chicago area with his wife and three kids.