Realtime publishers

Tips and Tricks Guide™ To

Windows Administration

Don Jones and Dan Sullivan

Copyright Statement

[**Editor's Note:** This eBook was downloaded from Realtime Nexus—The Digital Library for IT Professionals. All leading technology eBooks and guides from Realtime Publishers can be found at http://nexus.realtimepublishers.com.]

Tip, Trick, Technique 5: Evaluating Windows Server Backup

Windows Server Backup has been entirely rewritten for Win2008, and it's finally—after more than a decade of Windows' existence as a server operating system (OS)—a viable choice for many real-world backup and recovery tasks, especially in smaller environments. However, it's not a do-it-all solution; you should be prepared for significant disadvantages and weaknesses.

Installing Windows Server Backup

Like nearly every component of Win2008, Windows Server Backup (WSBackup) isn't installed by default. You'll need to open Server Manager, go to Features, and as shown in Figure 9, manually add the Windows Server Backup feature. It's a good idea to add the Command-line Tools sub-feature because you'll gain the ability to add backups to other automated processes in Windows PowerShell commands and scripts.

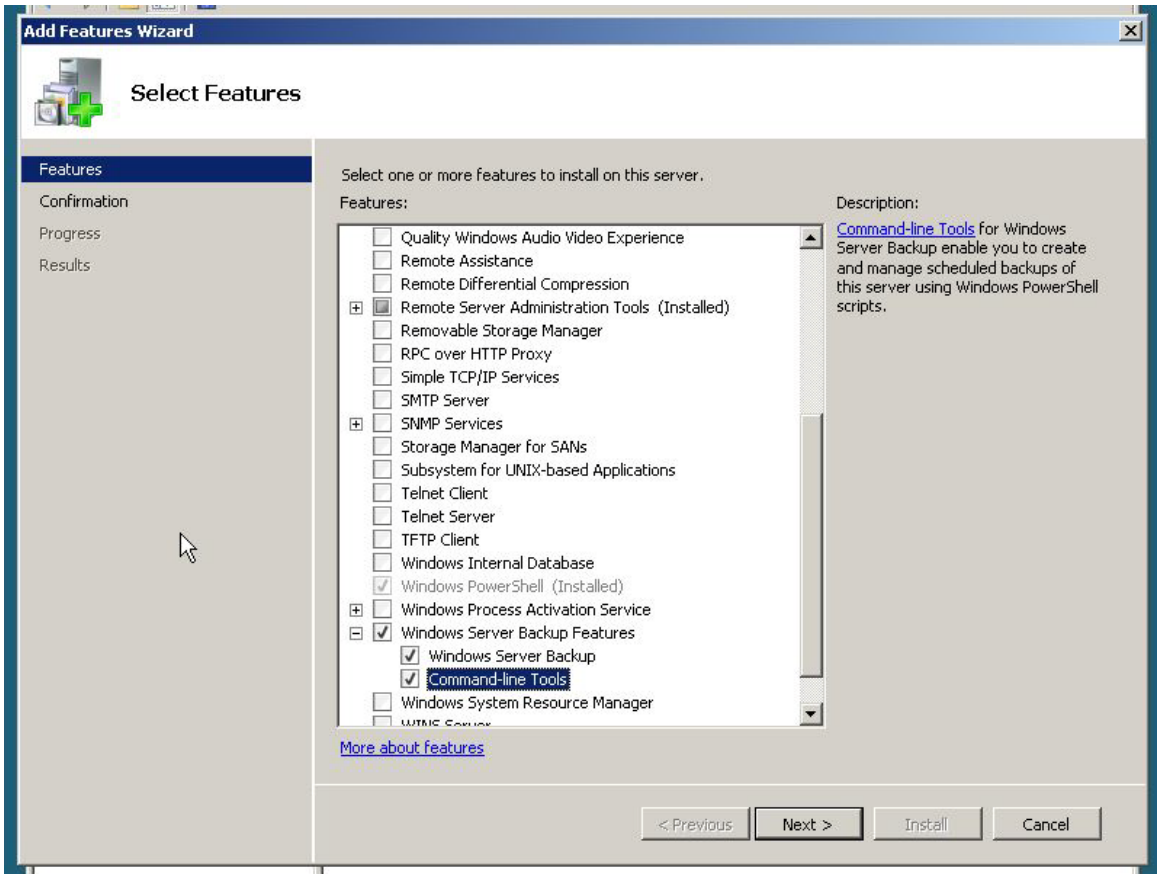

Figure 9: Adding Windows Server Backup.

Note

The need to add this feature can actually be a little confusing because Windows installs a shortcut on the Start menu for Windows Server Backup even if the feature itself isn't installed. Clicking the shortcut opens a console that tells you that you need to install the feature.

Using Windows Server Backup

Let's be perfectly clear in that WSBackup is intended to back up data and applications on the local computer; Microsoft doesn't position this feature as anything more than a very basic, local, bare-bones utility. Operations are primarily wizard-driven, such as the Backup Schedule Wizard that Figure 10 shows. With this wizard, you can select what you want to back up, when you want to back it up, where the backup will be stored (disk only—no tape support), and so on.

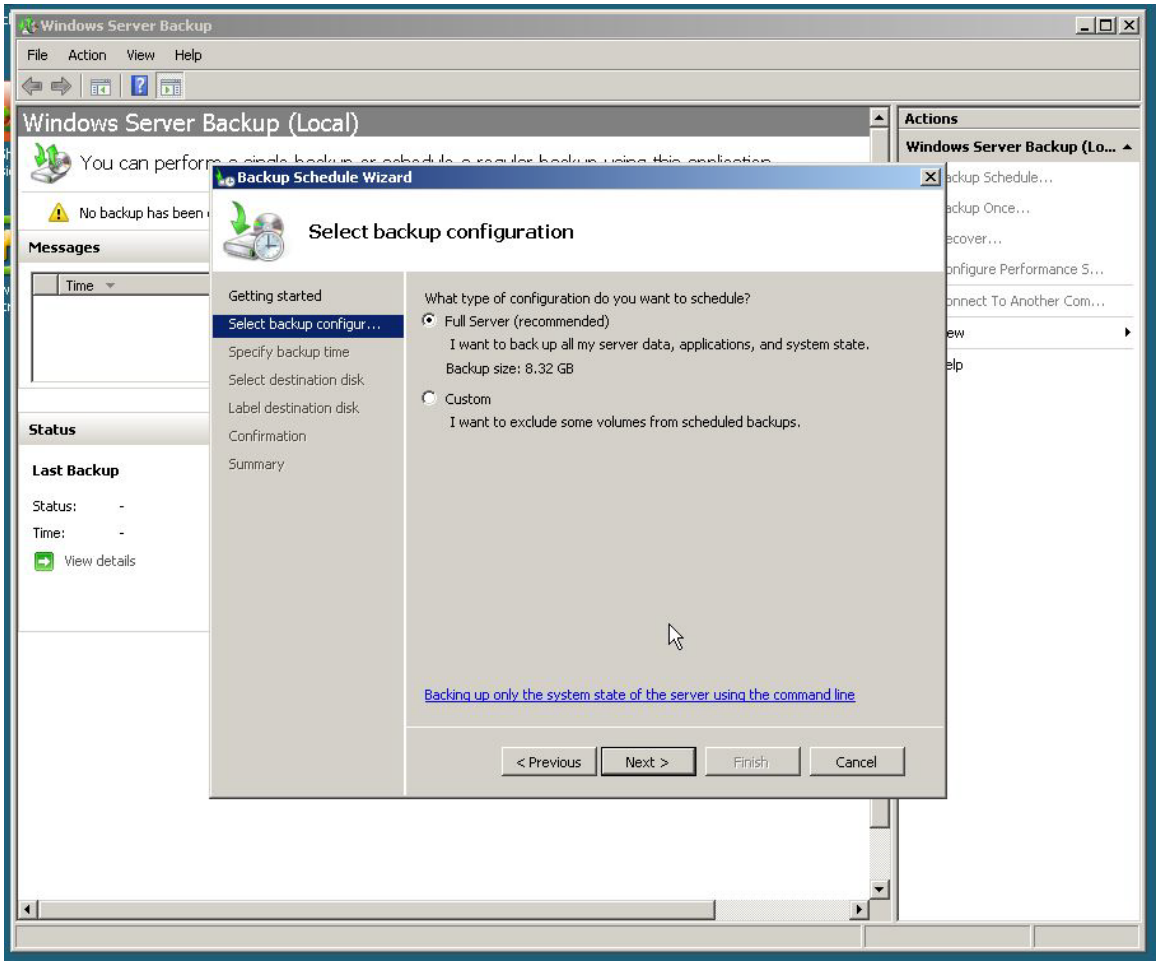

Figure 10: Configuring a backup.

You can restore a backup that was made from the local computer or from another computer (if you're trying to recover an entire system, for example, or need to grab a few files from a backup that was made of another computer). In addition, you can restore individual items from a backup as well as the entire thing.

As Figure 11 shows, you can configure backup performance by simply selecting the type of backup that will be made: a full backup (doesn't hit hard the server itself in terms of performance and cleans up Volume Shadow Copy files), or an incremental backup (leaves behind Windows' Volume Shadow Copy files and may diminish server performance somewhat). You can also make this decision on a per-volume basis.

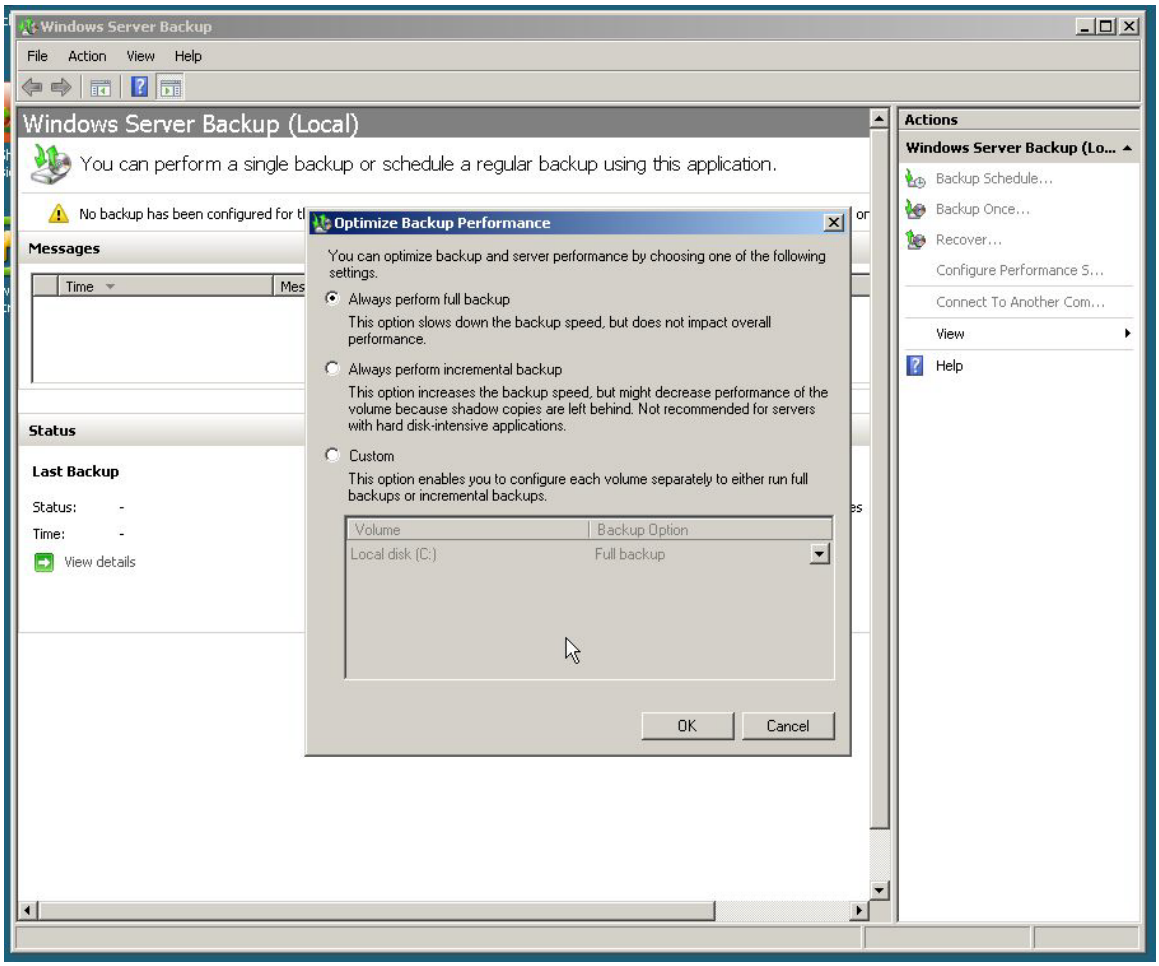

Figure 11: Configuring backup performance.

> **Note**
>
> Volume Shadow Copy (VSC) is designed to keep old versions of files handy in a disk-based store for easier recovery; users can use Windows' Previous Versions tab on a file's Properties dialog box to access VSC versions. Upon making a full backup, VSC files are normally cleared because the files protected by VSC are now safely in a backup.

Although WSBackup itself is designed to back up the local computer only, you can use the management console to connect to WSBackup running on other computers, allowing you to manage *their* local backup operations without having to physically log onto their consoles. Figure 12 shows this task in action.

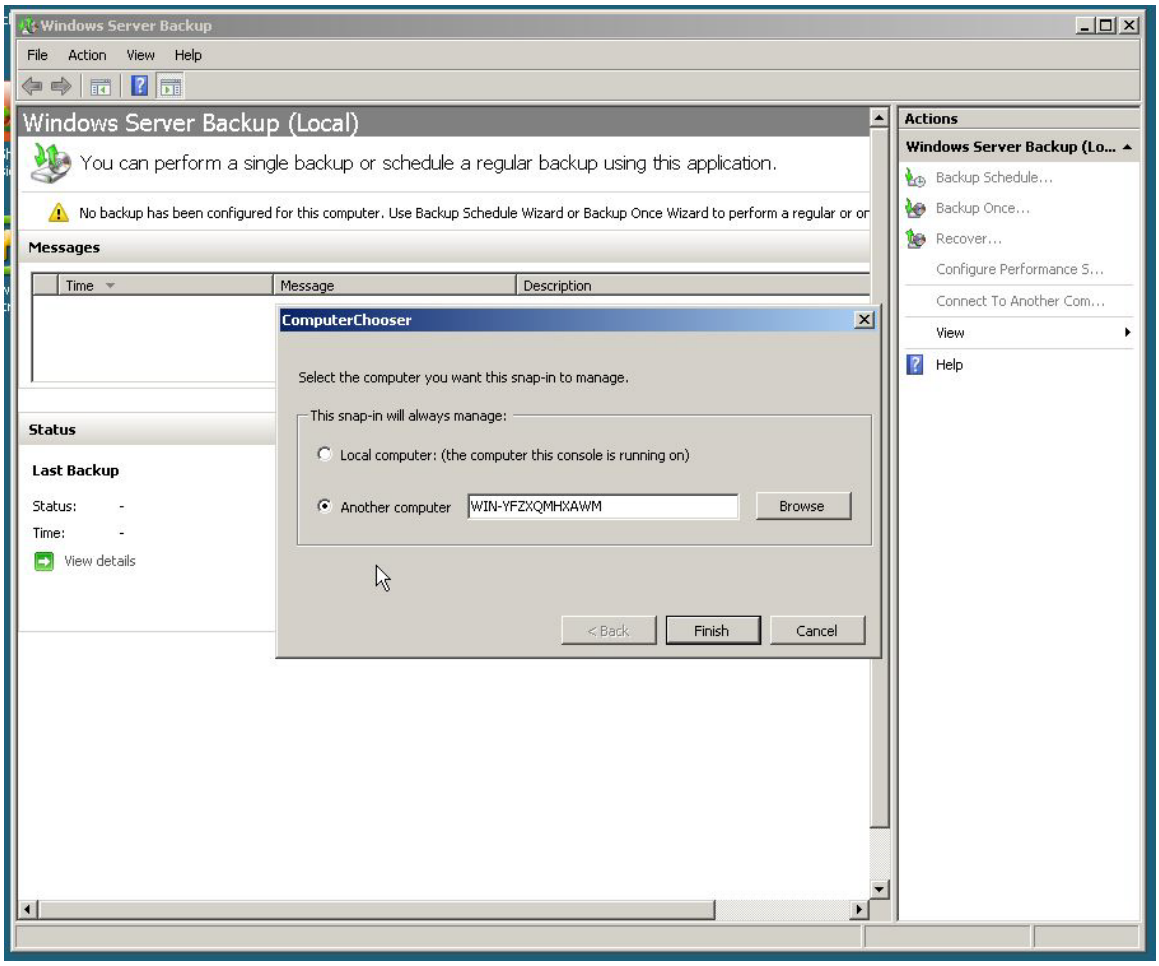

Figure 12: Connecting the WSBackup Console to another computer.

Pros and Cons

Most experienced administrators pretty much ignore Windows' built-in backup, and WSBackup isn't going to change their minds. For a very small environment dealing primarily with file and print servers, WSBackup is a reasonably effective, if bare-bones, means of making the backups you need to be safe. You'll need to move the backups off-server, of course, or they're at risk of a complete disk or system failure, and WSBackup doesn't make it easy to move those files around (it expects them to pretty much remain local). You can't save backups to any disk volume that contains Windows itself or application data, which means you'll need to install a dedicated volume—often not an option on a server that's already had all its disk space allocated.

Tip, Trick, Technique 6: Using Windows PowerShell

Although Windows PowerShell isn't specifically new in Win2008 (it was previously made available for Windows XP, Windows Server 2003, and Windows Vista), Win2008 is the first version of Windows that *includes* Windows PowerShell. A complete discussion on PowerShell is a book unto itself, but there are a few things you should be aware of and plan to take advantage of right away.

What Is Windows PowerShell?

Everyone who has heard of PowerShell has an idea of what it is: a command-line tool, a scripting language, or something. It's almost easier to explain what PowerShell *isn't:*

- It's not a scripting language. True, it does have scripting capabilities, but it's not quite the same as something like VBScript. It's more like the batch language in the old Cmd.exe shell but a bit more refined. It's simple—just 14 keywords in PowerShell v1—but it's very flexible and extensible.
- It's not a shell. Not, at least, in the same sense as Cmd.exe. PowerShell—the actual, under-the-hood guts of PowerShell—is an engine capable of running commands and scripts; the most common way for us human beings to tell it which commands to run is to type those commands into a *host* window, which is a command-line interface.

PowerShell is a standardized means for Microsoft to package administrative functionality. They're not quite command-line tools, although we humans can access them through a command-line interface. The big part there is *standardized* because PowerShell is the first time that Microsoft has created a clear, documented standard for exposing administrative functionality. PowerShell can be accessed from a command-line window, true, but it can also be hosted by graphical applications that run commands in the background. In some cases, you might be using a GUI console and not realize that PowerShell is actually doing all the work behind the scenes.

It's safe, in casual conversation, to refer to PowerShell as a command-line interface because that's how most of us will experience it directly.

Enabling Windows PowerShell

Although PowerShell is *included* with Win2008, it isn't installed by default: As Figure 13 shows, you have to enable its feature in order to start using it. Doing so will also enable the .NET Framework v3.0, which is the version that ships with PowerShell. PowerShell actually requires v2.0, which is a subset of v3.0.

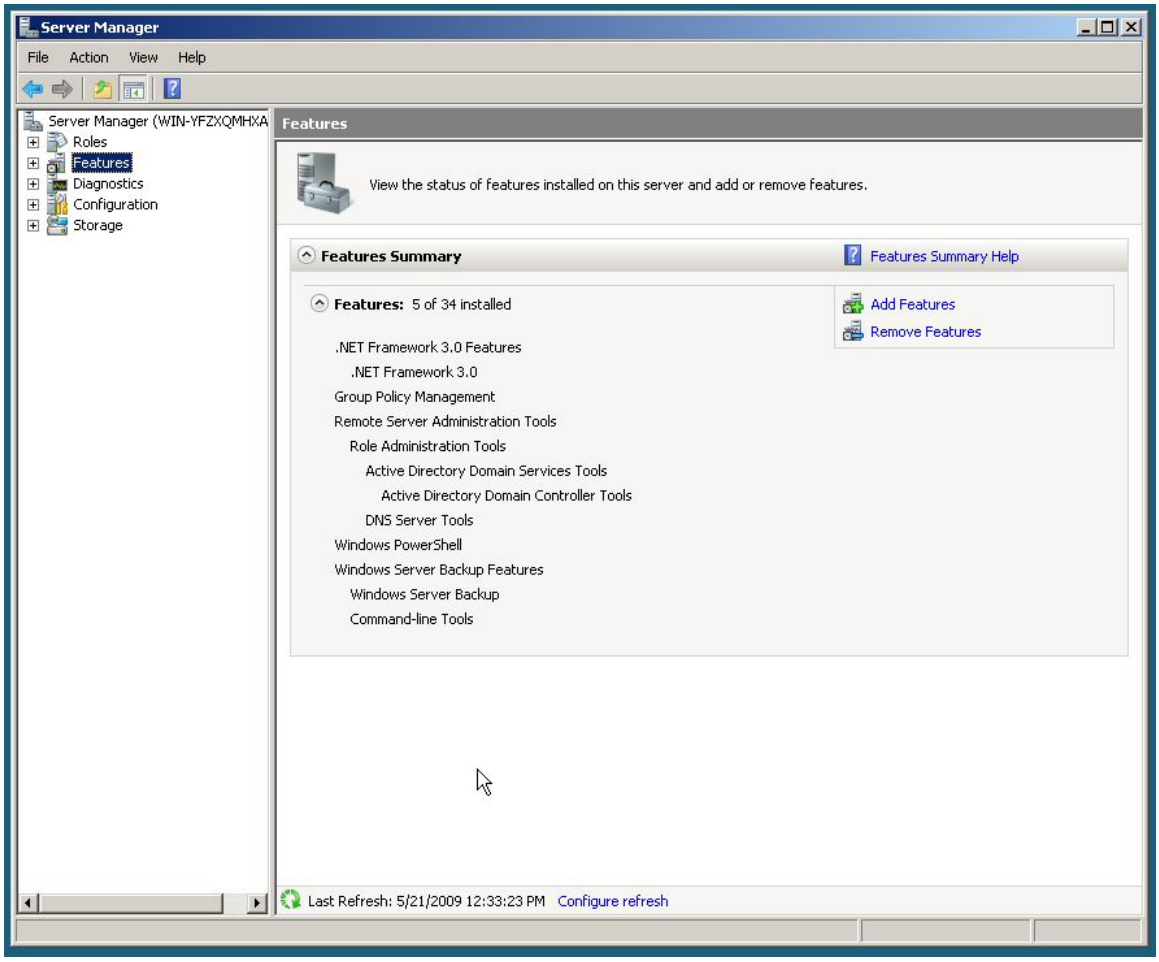

Figure 13: Windows PowerShell is an optional feature.

Note

The "R2" release of Win2008 actually does install Windows PowerShell v2 by default, which means the latest version of the Framework (3.5) is also installed by default. Because of PowerShell v2's new features, Microsoft feels—and I agree—that everyone will want and need PowerShell on every computer.

Windows PowerShell Security and Profiles

PowerShell has the ability to execute "script files," which are essentially a batch of commands executed in sequence, so Microsoft has obvious concerns about PowerShell and security. The last scripting language Microsoft pushed out, VBScript, was a dismal failure in terms of security, enabling mass virus attacks such as "I Love You," "Melissa," and other famous malware; the company certainly didn't want PowerShell landing in the same boat.

Understand that the potential danger in PowerShell does *not* come from running commands interactively. Typing a command, getting the syntax right, and doing anything requires a certain amount of expertise and isn't something you can typically trick someone into doing. In any event, no command will work unless the user has the necessary underlying permissions in the first place—PowerShell isn't a way to bypass Windows security. No, the real danger in PowerShell comes from scripts. That's because a script *is* something you can trick someone into running, and a script may contain entire sequences of commands that the tricked person might normally know not to run. Tricking an admin is especially deadly, because the admin will usually have permission to do all kinds of dangerous things.

So PowerShell's security focuses on script execution, primarily through a mechanism called the *execution policy*. By default, this policy is set to "Restricted," which prevents scripts from running entirely. Problem solved.

Changing the policy—by using the **Set-ExecutionPolicy** command within PowerShell itself—requires local Administrator privileges, as the setting is stored in the HKEY_LOCAL_MACHINE portion of the Windows registry. You can also control this setting centrally using a Group Policy administrative template that's available from http://download.microsoft.com (just punch in "PowerShell adm" in the search box to find the download). A Group Policy-applied setting overrides anything else.

So what might you change the policy to? "Unrestricted," the loosest setting, is stupid; you're putting PowerShell right back into the VBScript days, allowing any script to execute at any time. The next-higher setting, "RemoteSigned," might sound promising. It allows *local* scripts to execute without restriction but requires *remote* scripts—those downloaded from the Internet or accessed via UNC—to contain a digital signature. *This setting isn't any safer than* "Unrestricted," no matter what anyone tells you. I'll explain.

When a script is digitally signed (something you can accomplish using the **Set-AuthenticodeSignature** command), an encrypted copy of the script is added to the end of the script file in a special block of comments. When running the script, PowerShell decrypts this signature and compares it with the clear-text copy of the script. If the two match, the signature is "intact" and the script executes. If the signature doesn't match the script, the signature is "broken" and the script won't execute. This in and of itself doesn't prevent maliciousness, but here's what does: Obtaining the necessary digital certificate—a Class III Authenticode Code-Signing Certificate, to be specific—typically requires you to prove your identity, in some fashion, to the certificate issuer. Your identity becomes a part of the certificate and of any signatures you create using that certificate. Thus, if you create a malicious script, and sign it, PowerShell will run it—and anyone affected by it will be able to divine your identity and hunt you down. So, in very general terms, signed script = safe script.

Note

This "safe" depends entirely on the certificate issuer doing a good job of actually checking your identity when issuing a certificate. You can configure Windows to trust certificate issuers who you believe do a good job; you can configure it to *not* trust issuers who you don't believe do a good job. If PowerShell encounters a signature that came from an untrusted issuer, the signature and the script are also considered untrusted and the script won't run.

So, imagine a scenario: Your computer gets infected with a piece of malware. Only, rather than trying to do anything nasty, it just modifies an innocent little text file on your computer. One with, say, a ".ps1" filename extension—a PowerShell script, in other words, that you've already written. The next time you go to run this *local script,* the commands added by the malware also execute—and chaos ensues.

Or, even worse, the malware creates a simple text file with the name profile.ps1, inside a folder named "WindowsPowerShell," right in your Documents folder. No big deal, right? Wrong: This is a PowerShell *profile* script, and it is going to execute *automatically* the next time you open the shell! Worse, this file doesn't exist by default, so it's easy for a piece of malware to create it without you knowing. User Account Control (UAC) won't save you here because it's just a simple text file in your Documents folder—nothing you need Administrator privileges to access.

The solution? PowerShell's third execution policy, "AllSigned." This setting requires all scripts to carry a signature, created by using a certificate that came from a trusted issuer. Create your own profile script (a blank one is fine) and sign it to prevent a piece of malware from plopping down a profile script, and you're protected. Sure, you have to sign your scripts before they'll run—no big deal. The better commercial script editors (PrimalScript and PowerShell Plus Professional Edition come to mind) will do that automatically for you, if you want them to. Don't want to buy a certificate? Run **help about_signing** in PowerShell and read how to use the **MakeCert** utility to create a free, local-use-only certificate for your own scripts.

Using Windows PowerShell: The Basics

As a command-line shell, PowerShell works a lot like the Cmd.exe shell you're probably familiar with: type a command, add on any necessary parameters, and you're ready to hit Enter. Need to try again? Hit the up arrow, modify the command, hit Enter, and you're done.

So how do you get around your system in PowerShell? If you've ever navigated a disk drive in Cmd.exe, then you know how to do it in PowerShell.

Type Dir to get a listing of files and folders—or type Ls, if you prefer that. Cd will change folders. Del will delete files, so will Rm. Type will display the contents of a text file, as will Cat. A backslash is a path separator, as is a forward slash. So whether you're comfortable with UNIX- or DOS-style syntax, you're good to go.

There are some caveats. "cd.." won't go up one folder level; you need to use "cd .." with a space. That's because PowerShell assigns a special meaning to the space character: it's a separator between a command and its arguments. That's why Cd c:\program files doesn't work—the space between "program" and "files" confuses it. Add quotes—either single or double—and run Cd "c:\program files" instead. That's pretty much what you would do in Cmd.exe or even most UNIX shells, by the way.

So there you have it: A completely arbitrary (like "Ls" is intuitive?) set of commands that you've probably already memorized and can use to navigate through a hierarchical database. Yes, a database—that's what the file system really is, after all. It's not relational like an Access or SQL Server database, but it's hierarchical, not unlike an Exchange Server mail store, or the Windows registry, or even Active Directory (AD). Speaking of which, would you like to learn a whole new set of commands that let you navigate the registry or even AD?

I hope you said "no" because who wants to learn a whole new set of commands when YOU ALREADY KNOW A SET that should do the job? In other words, why can't we just run "Cd HKCU:" to change into the HKEY_CURRENT_USER registry hive? Why can't we run "Ls" to get a list of registry keys? Run "Cd Software" to change into that key, and "Del *" to delete everything—whoops.

Well, it turns out you can in PowerShell. Try it. That's because PowerShell has little adapters called PSDrives that allow PowerShell to see different forms of storage as if they were disk drives. The Certificate Store, environment variables, registry, and more are just the beginning. More PSDrive adapters can be added in, and products like SQL Server 2008, AD (in Win2008R2), and others do just that. Run "Get-PSDrive" to see a list of all the drives currently available, and use "New-PSDrive" to create new drive mappings (remember, these are PS Drives, so they only live in PowerShell—you won't see them in Windows Explorer).

PSDrives illustrate a key part of PowerShell's design philosophy: Take ONE set of skills—preferably a skill that administrators already have—and leverage it as widely as possible. That means less learning for you while expanding the number of things you can do. It's like the moving walkway at the airport: slow people are supposed to keep right so that faster people can pass on the left. It's the same skill that we're supposed to use on the highway, leveraged in a new location. Sadly, most people seem to lack the skill in either scenario, but you get the idea. And even that makes a good point: PowerShell is leveraging skills that administrators SHOULD already have; if you've stayed away from any kind of command-line administration, you have done yourself a disservice because PowerShell assumes you've worked at least a little from the command line. If you haven't, PowerShell won't be impossible to use, but it will be a bit more of a learning curve because you lack some of the background experience that PowerShell is trying to leverage to make things easier on you.

Let THAT be a lesson for you. A big reason to learn PowerShell NOW is because there will be future versions that add MORE functionality. By learning PowerShell NOW, you can start gaining the background experience that will make future versions more incremental and easier to learn; the longer you wait, the harder it will be to learn each successive version.

All About Commands, Aliases, and Parameters

So all of these things we ran—CD, DIR, LS, and whatnot—are all "commands." Technically, because they're within PowerShell, they're called "cmdlets" (pronounced "command-lets," not "cee-em-dee-lets"). Actually, technically, what we've been using so far are aliases.

Let me back up a bit.

PowerShell's functionality comes primarily from these cmdlets, all of which are written by developers working in a .NET language such as C# or Visual Basic. Cmdlets come packaged in a *snap-in*, which is basically a DLL file. You can think of them as similar to the snap-ins used by the Microsoft Management Console (MMC), in that they add product-specific functionality to an otherwise empty shell or console.

Cmdlets use a consistent naming scheme devised by Microsoft. Cmdlet names consist of a verb, such as Get, a dash, and then a singular noun, such as Service (for example, Get-Service). The list of verbs is actually fairly short and is intended to be used consistently. Changing something uses the Set verb, so you have cmdlets such as Set-Service and Set-ExecutionPolicy—never Change-Policy or Configure-Service. Using consistent verbs helps folks like us guess the right command name without having to pore through manuals. For example, based solely on what I've written here, can you guess the Exchange Server command that would retrieve user mailboxes? Get-Mailbox.

The downside of these command names is that they can be long. Not that long is inherently bad—long also means clearer and easier to remember. But long does mean harder to type, and nobody wants that. So PowerShell has a system for aliases, which are simply nicknames for a command. Dir is an alias for Get-ChildItem, Type is an alias for Get-Content, Ps is an alias for Get-Process, and so forth. The alias is simply a way of shortening the command name or making the cmdlet name look like a familiar command (such as Dir or Del). The alias doesn't change anything about the way the cmdlet works. Run Dir /s and you'll see. That generates an error because the Get-ChildItem cmdlet, which is what's really being run when you type Dir, doesn't support a /s parameter.

Which brings us to parameters, I suppose. In sticking with the "consistency" theme, PowerShell finally brings us a consistent command-line syntax for parameters. Parameters always begin with a dash—not a slash—and the parameter names are really clear: -computerName, -path, -filter, -exclude, -credential, and so forth. The parameter name is followed by a space and then whatever value goes with the parameter, if appropriate. A parameter such as –append wouldn't usually take a value; it's just a switch, telling the cmdlet to append content to existing content. A parameter such as –computerName obviously does need a value—the computer name you want to pass along. So that's why Dir /s doesn't work: the Get-ChildItem command doesn't recognize /s as a parameter. Actually, it'll think it's supposed to be a path because PowerShell uses both / and \ as path separators. However, the command does have a –recurse parameter that'll do what you want.

There's no way to create an alias so that "Dir /s" behaves as "Get-ChildItem –recurse"—aliases are nicknames only for command names, not for anything else, and not for any parameters. Using an alias doesn't change the command syntax in any way; you're simply substituting a shorter name for the command, nothing more.

That said, you don't have to type the whole parameter name—honestly, typing –computerName all the time would be a hassle. You only have to type as much of the parameter name as needed to distinguish it from other parameters. So, for Get-ChildItem, instead of typing –recurse, you could type –r because there are no other parameters of that command that begin with "r." The "r" alone is enough to let the shell figure out which command you meant. In other cases, a few more letters may be needed: I usually type "-comp" for –computerName, for example. It's probably more letters than I technically have to type in most cases, but it's enough to help me visually determine what parameter I meant.

And there's always **Help:** PowerShell's built-in help system even accepts wildcards, so running **Help *Service*** will help you find all the commands related to services, while running something like **Help Get-WmiObject** will offer complete help for that entire command and all its parameters. In PowerShell v2 (with Win2008 R2), the **Help** command picks up an **–online** parameter, which pops up the latest and most accurate help in a Web browser, straight from Microsoft's Web site.

Tip, Trick, Technique 7: Understanding Hyper-V

Hyper-V is an exciting new feature of Windows Server 2008. Although much has been, and will be for some time to come, written on Hyper-V and its major competitors—VMware vSphere (ESX Server), and Citrix Xen Server—it's important to understand what Hyper-V is and isn't because it comes with Win2008.

Hyper-V, Hypervisor—What's it All Mean?

Hyper-V is Microsoft's brand name for their Windows-based hypervisor. A *hypervisor* is a special type of software that's specifically designed to enable virtualization: the ability for one computer to effectively mimic the operation of many "virtual" computers at the same time. The hypervisor installs on a *host* computer and has direct (more or less) access to its hardware; it then enables one or more *virtual machines* to execute in memory. Each virtual machine, or *guest,* can run its own operating system (OS)—which need not be Windows—and each guest OS thinks it's running on its own dedicated hardware.

Hyper-V is technically a *type 1* hypervisor, meaning the hypervisor itself runs on "bare metal," or directly on the server's hardware. Win2008 automatically creates a special virtual machine where the rest of Win2008 is installed. So, when you're using a Win2008 machine that has Hyper-V installed, you're always running at least one virtual machine—the one that Win2008 itself is running on. That "primary" virtual machine is the one that gets to tell the hypervisor what to do. It's not quite a guest virtual machine because it does have a special management relationship to the underlying hypervisor.

How Does Hyper-V Licensing Work?

You need to own a Win2008 license to run Hyper-V. Beyond that, you'll also need licenses for whatever guest OSs you plan to run inside your virtual machines. The free, downloadable "Windows Hyper-V Server" product doesn't include licenses for anything but Hyper-V itself; any guest OSs will need a license.

When you buy a copy of Win2008, however, it comes with a certain number of licenses for guest virtual machines running copies of Win2008. The Datacenter edition of Win2008, for example, lets you run an unlimited number of virtual machines that run any other editions of Win2008; Win2008's Enterprise edition includes guest licenses for up to four Win2008 guests.

Is Hyper-V a "Bare Metal" Hypervisor?

Yes. Lots of people like to argue this because when you install Hyper-V, you appear to be using a full copy of Windows. So, they argue, if Hyper-V requires Windows, it's technically a *type 2* hypervisor, meaning the hypervisor doesn't talk directly to the hardware. This was the case with the predecessor, Microsoft Virtual Server. Its architecture looked a bit like what's shown in Figure 14, with the hypervisor clearly running atop Windows and depending on Windows to provide access to the hardware. Here, the hypervisor runs as an application, at the same level as something like Exchange Server.

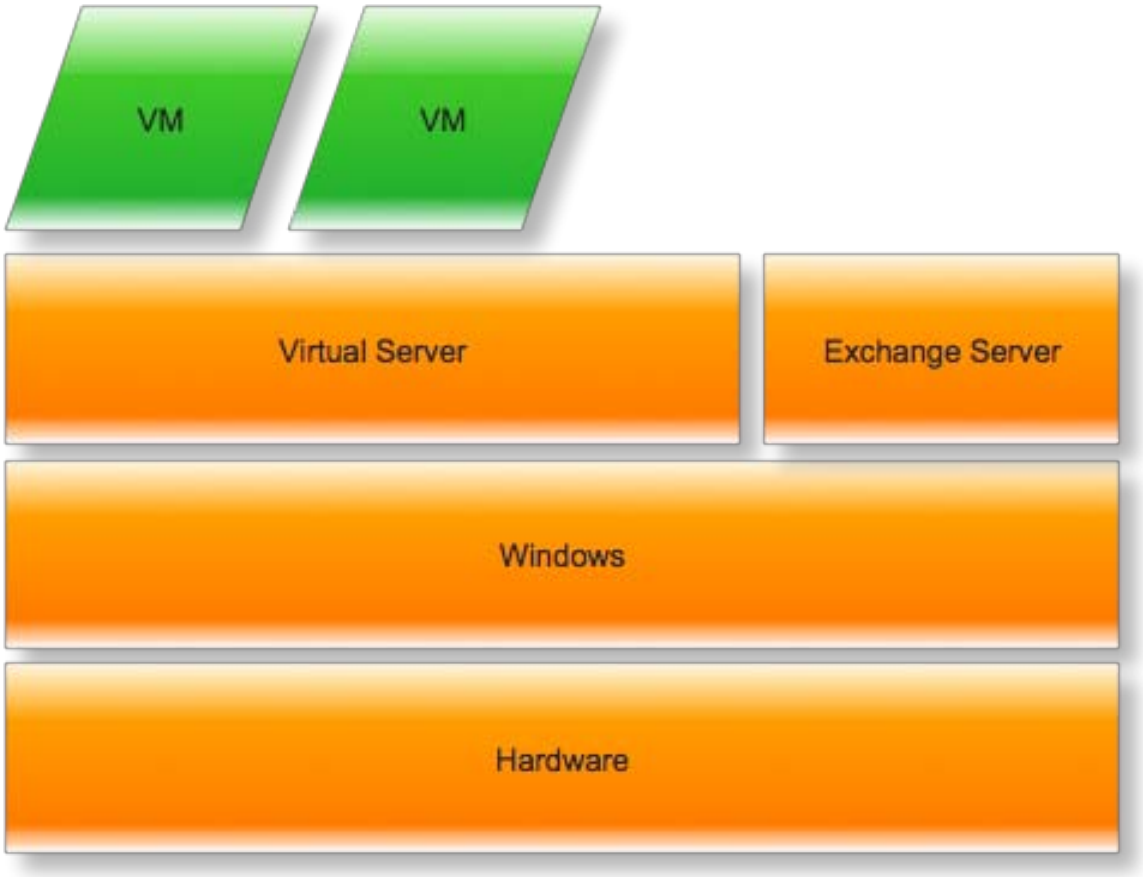

Figure 14: A type 2 hypervisor.

Hyper-V's architecture is shown in Figure 15. What fools folks about Hyper-V is that it *always* installs a virtual machine—technically, a *partition*, to use Microsoft's terminology—containing a full Win2008 install. So you *always* see Windows, even though Hyper-V itself isn't talking through Windows to get to the hardware.

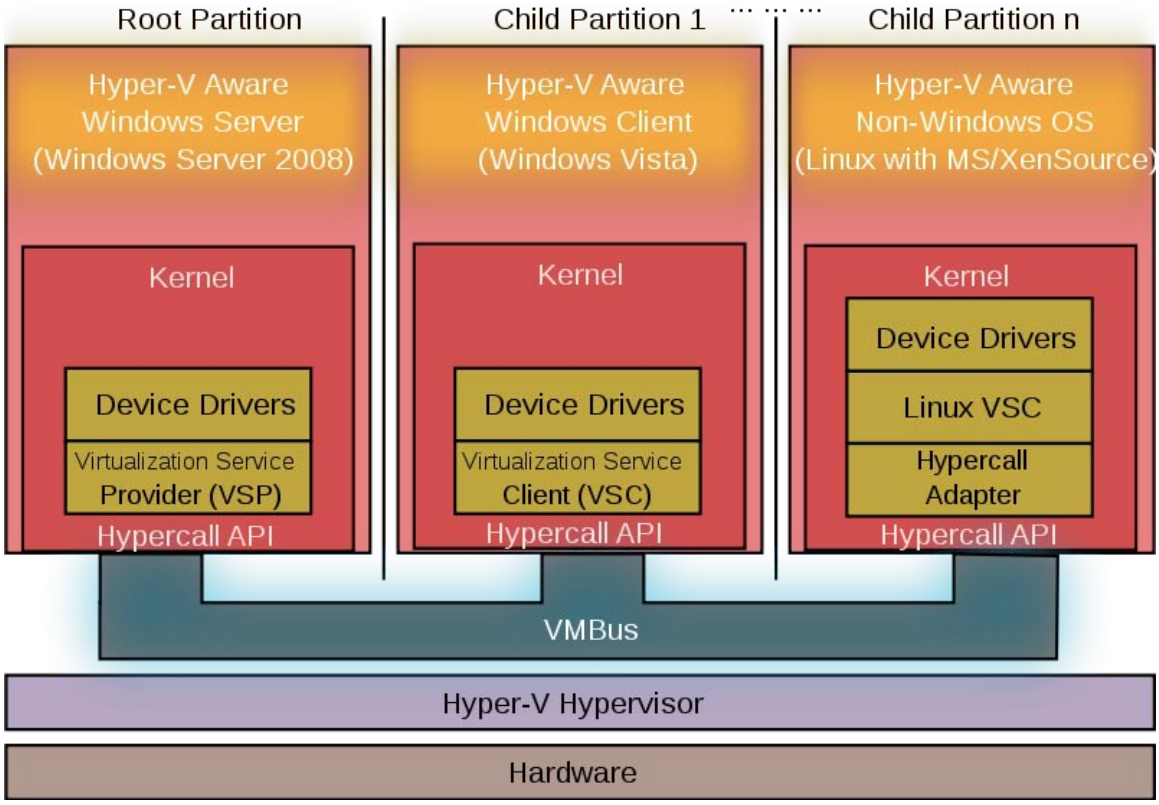

Figure 15: Hyper-V architecture.

Also shown are some unique features of Hyper-V, such as the ability of OSs that know about Hyper-V to realize that they're running in a guest virtual machine. This lets them feed specific types of information (such as performance) to the host for better manageability, and lets Hyper-V communicate with the guest OS to perform key tasks, such as better managing shutdowns. Non-aware guest OSs can also run but get fewer manageability improvements.

In fact, there is a way to run Hyper-V without running the full copy of Windows: Windows Server Core. The free "Windows Hyper-V Server" downloadable product uses this, and you can set up one yourself. It simply installs Server Core into the "root" partition so that you get a smaller Windows footprint in the root and more resources freed up for running your other partition.

Download Additional eBooks from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this eBook to be informative, we encourage you to download more of our industry-leading technology eBooks and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.