



Comparing Bash with the Windows Vista shell

SHELL GAMES

Microsoft's new PowerShell relies on .NET framework libraries and thus has access to a treasure trove of functions and objects. How does PowerShell measure up to traditional shells like Bash?

BY MARCUS NASAREK

Both Bash and the Windows Vista PowerShell include commands for navigating directories, managing files, and launching other programs. System administration is an important duty for the shell, and Bash and PowerShell are equipped to help manage systems from the command prompt.

Whereas Bash typically relies on a combination of newer tools and classic Unix utilities, the PowerShell has its own set of command-line programs. Windows refers to PowerShell commands as cmdlets. The PowerShell cmdlet called *Get-Process* is a counterpart to *ps*, and the cmdlet *Get-Content* corresponds to *less*. PowerShell differs significantly from previous Windows command shells. In this article, I look at how Windows Vista PowerShell compares with Bash.

To support program control, a shell needs elements for conditional execution. *for* or *while* evaluate a variable to support a defined number of iterations.

Listing 1 compares outputting a counter with *for* in PowerShell and Bash.

Bash and PowerShell are similar with respect to case-based program flow control with *if* or *switch*. The definition of functions, the use of environmental variables, scoping (restricted validity of variables), the use of regular expressions, and the evaluation of program return values are all similar in both shells.

Restricted Launch Permission

An initial difference between PowerShell and Bash occurs when a script is executed. PowerShell will not launch script files by default and thus only supports interactive use. However, PowerShell will run scripts if they are digitally signed. The digital signature identifies the script's author because the author is the only person capable of creating the signature by cryptographic means. Accepting the author's signature certificate

means that you trust the author. PowerShell will not run "unknown" scripts at all. To run a script without a signature, you need to change the execution policy to *RemoteSigned* at the command line like so:

```
PS:> Set-ExecutionPolicy RemoteSigned
```

Listing 1: Loops

A for loop in the Windows Vista PowerShell:

```
PS:> for ($i=1;$i -le 3;$i++) {
Write-Host $i }
```

```
1
```

```
2
```

```
3
```

```
PS:>
```

The same loop in Bash:

```
bash[~]$ for ((i=1;i<=3;i++)); do
echo $i; done
```

```
1
```

```
2
```

```
3
```

```
bash[~]$
```

Listing 2: Get-Member Output

```

01 PS:> Get-Content Textdatei.txt | Get-Member
02
03     TypeName: System.String
04
05 Name          MemberType      Definition
06 ----          -
07 Clone         Method             System.Object Clone()
08 CompareTo     Method             System.Int32 CompareTo(Object
    value), System.Int32 CompareTo(String strB)
09 Contains      Method             System.Boolean Contains(String
    value)
10 CopyTo       Method             System.Void CopyTo(Int32
    sourceIndex, Char[] destination, Int32 destinationIn...
11 ...
12 Length       Property
13
14 PS:>

```

This command tells PowerShell to accept all local scripts. If you downloaded the data, or if the data is attached to an email, the shell will continue to insist on a signature. The ability to prevent commands from external sources from executing is a whole new league security-wise, and scripted viruses such as “Love Letter” [1] lose their threat.

In contrast to this, Bash does not rely on digital signatures to evaluate a script’s execution permissions. Instead, filesystem permissions determine whether the script is permitted to run.

Both shells share some surprising common ground in the way they handle system configuration. New to the world of Windows is the way PowerShell re-

gards everything as a filesystem and not only navigates the filesystem and drives, but the Registry, the certificate store, and environmental variables. You can copy, rename, and move Registry values just as you would files on a drive. The PowerShell refers to these virtual filesystems as *Providers*, thus implementing a philosophy that Linux has always offered to Bash: “Everything is a file.”

Forwarding

PowerShell’s most powerful tool is the pipe, which supports ordered passing of values, allowing the use of output from one command as input to the next command. Bash supports pipes too, but it does not make particular demands on

input and output files. Bash trusts that the next command in the chain can do something meaningful with the output.

In PowerShell, all cmdlets create defined objects as their output data instead of text-only output. Even if it appears that only strings are passed in text output, what happens is that the complete output is converted to an object.

Objects can be queried with the *Get-Member* command, which outputs the elements and functions of the object. See Listing 2. For example, the command

```

PS:> Get-Content Textdatei.txt |
Sort { $_.Length }

```

lets you sort the lines in a text file by length with the object’s *Length* property.

Although passing data objects is slightly more complex, this object orientation helps standardize operations and supports handling of complex data structures. Bash cannot compete here; instead, it relies on the abilities of external programs to handle data structures.

For example, Bash needs an external XML parser (like Saxon or Xalan-J) to parse XML files. Listing 3 is a short PowerShell script that loads an RSS feed off the Internet in the form of an XML file. The script defines a *Show-SpiegelRSS* function that gets the current RSS Feeds.

Conclusions

In one respect, PowerShell relies on the Unix concept that many small utilities are preferable to one large, custom-made utility. At the same time, it adopts an object-oriented approach that makes larger scale projects simpler at the cost of a steeper learning curve. The major problem with objects is that you need to invest significant time in discovering which function or object you need. The *Get-Member* cmdlet is likely to see much use in PowerShell.

Bash is useful as a plain but straightforward tool for most daily tasks. If it comes to the need for advanced uses and complex data structures, you can branch out into object-oriented Python or the graphical capabilities of Tcl/Tk. ■

Listing 3: Show-SpiegelRSS.ps1

```

01 # Show-RSS.ps1
02 # Declaration of variables for URL
03     $feed="http://rss.news.yahoo.com/rss/linux")
04     Write-Host -ForegroundColor "green" "RSS-Feed: " $feed
05 # Download RSS feed
06     $wco = New-Object System.Net.WebClient
07     $rss = [xml]$wco.DownloadString($feed)
08 # Show title
09     Write-Host -ForegroundColor "red" $rss.rss.channel.title
10 # Display short form
11     $rss.rss.item | Select-Object title,description | format-table
12 # Display title and description of entries
13     $rss.rss.channel.item | Select-Object title,pubDate,description
    | format-list

```

INFO

[1] CERT-Advisory CA-2000-04 “Love Letter Worm”: <http://www.cert.org/advisories/CA-2000-04.html>