PowerShell Scripting Best Practices

This document is intended as guidance for how all PowerShell scripts should be written for security, functionality, and consistency.

To help enforce these rules during your dev process, consider using PSScriptAnalyzer. Each of the sections below contains the rule code for enabling the corresponding check in PSScriptAnalyzer where one exists. Additionally, the analyzer is built in as a linter in the PowerShell VSCode extension. We have created a settings file to be used with this linter.

Do not use Invoke-Expression or script blocks built with string concatenation

PSScriptAnalyzer Rule: AvoidUsingInvokeExpression

Invoke-Expression and non-parameterized script blocks are both vulnerable to injection.

Invoke-Expression injection

Invoke-Expression executes a specified string, which is vulnerable to injection attacks. As an example:

```
$function = "Write-Host"
$argument = "hello; Write-Host injected"

Invoke-Expression "$function $argument"
```

will return:

```
hello
injected
```

While the following script:

```
$function = "Write-Host"
$argument = "hello; Write-Host injected"

& $function $argument
```

will return:

```
hello; Write-Host injected
```

as would be expected.

In general, this guidance can be summarized as **don't use Invoke-Expression**.

Script-block injection

Similarly, the following script block is also vulnerable to injection.

```powershell
$UserInputVar = "hello; Write-Host injected"
$DynamicScript = "Write-Host $UserInputVar"
$ScriptBlock = [ScriptBlock]::Create($DynamicScript)
Invoke-Command $ScriptBlock
```

This returns:

```
hello
injected
```

While this script:

```powershell
$UserInputVar = "hello; Write-Host injected"
[ScriptBlock]$ScriptBlock = {
        Param($SafeUserInput)
        Write-Host $SafeUserInput
}
Invoke-Command -ScriptBlock $ScriptBlock -ArgumentList @($UserInputVar)
```

correctly outputs:

```
hello; Write-Host injected
```

In general, this guidance can be summarized as **don't use [ScriptBlock]::Create**.

Prefix script and executable calls with &

When a script/executable is prefixed with an ampersand (`&`), the command which follows can be in quotation marks or include variables. This is not the case when the ampersand is not included. Thus, we recommend always including an ampersand.

Check $LASTEXITCODE after calling

Relatedly, `$LASTEXITCODE` should always be checked after running an executable to ensure that the script fails (or at least responds appropriately) to the executable failing.

Combining this with the previous piece of advice, the way to call `git add .` from a script would be:

```powershell
& git add .
if ($LASTEXITCODE -ne 0) {
  # behavior in case of error
}
```

To set it in a PowerShell script, e.g. to ensure a known value at the start of a script, reference the variable in the global scope.

```powershell
$global:LASTEXITCODE = 0
```

Set StrictMode and ErrorActionPreference at the top of every file

Scripts should always include the following just below the parameter definition block:

```
Set-StrictMode -Version 2.0
$ErrorActionPreference = "Stop"
```

This will ensure PowerShell uses the proper version and that encountered errors cause the script to fail.

Do not use aliases in scripts

PSScriptAnalyzer Rule: AvoidUsingCmdletAliases

Cmdlet aliases (such as `ls` for `Get-ChildItem` and `echo` for `Write-Output`) are not universal across all machines and all installs of PowerShell. Furthermore, aliases can cause confusion as the cmdlets frequently behave entirely differently from the commands the aliases are named for, e.g. cmd's `dir` vs. `Get-ChildItem` or bash's `wget` and `curl` vs. `Invoke-WebRequest`. Always use the actual cmdlet name.

To determine what cmdlet an alias points to, simply run:

Exceptions: `%` and `?` are aliases for `ForEach-Object` and `Where-Object`, respectively. These aliases can be left in code and warnings related to them can be ignored.

```
Get-Alias $alias
```

e.g.

```
Get-Alias ls
```

returns:

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           ls -> Get-ChildItem
```

Use CIM cmdlets rather than WMI ones

PSScriptAnalyzer Rule: AvoidUsingWMICmdlet

PowerShell recommends avoiding all the WMI cmdlets (`Get-WmiObject`, `Remove-WmiObject`, `Invoke-WmiObject`, `Register-WmiEvent`, `Set-WmiInstance`) and instead using the CIM ones (respectively, `Get-CimInstance`, `Remove-CimInstance`, `Invoke-CimMethod`, `Register-CimIndicationEvent`, `Set-CimInstance`).

Disable positional binding for your parameters

PSScriptAnalyzer Rule: AvoidUsingPositionalParameters

Positional parameters cause problems for code maintenance, as adding new parameters later down the line can break previous invocations. Instead, parameters should always be called explicitly. Setting:

```
[CmdletBinding(PositionalBinding=$false)]
```

will force this behavior.

Note: This rule will check for the usage of positional parameters rather than forcing binding to turn them off.

Was this helpful?