Les runspaces contraints

Par Laurent Dardenne, le 18/03/2017.

Version 1.0

Niveau

Débutant	Avancé	Confirmé

Conçu avec Powershell v5.1 Windows 7 64 bits.

Site de l'auteur : http://laurent-dardenne.developpez.com/

Chapitres

1 L'objectif

Chaque code Powershell est exécuté dans un runspace qui par défaut autorise toutes les commandes et mot-clé du langage.

Dans le cas où l'on souhaite que notre propre code exécute du code utilisateur on peut vouloir restreindre les commandes ou les opérations afin de limiter les accès aux ressources ou contrôler la cohérence du traitement soumis. Par exemple ne pas soumettre du code de suppression d'objet dans un traitement d'ajout d'objet.

Il y a quelques similitudes avec la notion de restriction de portée, de typage ou de validation à l'aide d'attributs :

```
Param(
   [ValidateRange(8,14)]
   [Int]$Test
)
```

Lorsqu'on applique ce principe à un runspace on restreint l'environnement d'exécution dans son ensemble et pas seulement un élément particulier d'un script.

2 Le principe

Pour le visualiser rapidement utilisons la section DATA d'un script qui est dédiée à l'internationalisation de scripts :

```
$File='C:\Temp\DataSection1.ps1'
@'
Data {
  if ($PSVersionTable.PSVersion -eq ([Version]"2.0.0.0"))
  {"PowerShell 2.0"}
  else
  {"PowerShell >= 3.0 "}

  Get-ChildItem C:\
}
'@ > $File

. $File
```

L'exécution du code contenu dans la section DATA provoque ces exceptions :

```
Property references are not allowed in restricted language mode or a
Data section.

…
A variable that cannot be referenced in restricted language mode or a
Data section is being referenced. Variables that can be referenced
include the following: $PSCulture, $PSUICulture, $true, $false, $null.

…
The type version is not allowed in restricted language mode or a Data
section.

…
The command ' Get-ChildItem' is not allowed in restricted language mode
or a Data section.
```

Ce qui ici est normal car cette section ne supporte par défaut qu'une seule commande (*ConvertFrom-StringData*), l'accès à un sous-ensemble de variables (celles indiquées dans le message d'erreur) et certaines opérations du langage (ici l'accès à une propriété d'un objet n'est pas possible).

La section DATA permet l'ajout de commandes via le paramètre ***–SupportedCommand*** :

```
$File='C:\Temp\DataSection2.ps1'
@'
Function Get-PSversion {$PSVersionTable.PSVersion }

Data -supportedCommand Get-PSversion {
  if ("2.0" -eq (Get-PSversion))
  {"PowerShell 2.0"}
  else
  {"PowerShell >= 3.0 "}
}
'@ > $File

. $File
PowerShell >= 3.0
```

Le cmdlet ou la fonction précisée devrait ne générer que des données utilisé dans la section DATA.

Pour cette démo, une conversion implicite évite l'usage du type [Version].

Maintenant affichons les informations du runspace :

```
$File='C:\Temp\DataSection3.ps1'
@'
Function Get-RSInfo {
 $ExecutionContext.host.Runspace.InitialSessionState|
  Select-Object *
}

Data -supportedCommand Get-RSInfo {
  Get-RSInfo
}
'@ > $File
. $File
LanguageMode                  : FullLanguage
…
```

Ici la propriété *LanguageMode* nous indique que le runspace n'est pas limité, ce qui peut nous surprendre dans un premier temps, mais c'est tout à fait normal car seul le code de la section *Data* est parsé par une classe spécifique limitant les possibilités du langage.

On remarque également que le code de la fonction *Get-RSInfo* n'est pas concerné par cette analyse, par contre de placer sa déclaration dans la section **Data** provoquera des exceptions :

```
Function declarations are not allowed in restricted language mode or a
Data section.
Script block literals are not allowed in restricted language mode or a
Data section.
Property references are not allowed in restricted language mode or a
Data section.
```

Si le paramètre ***-supportedCommand*** référence une commande d'un module celui-ci est chargé dans la session courante.

2.1 CheckRestrictedLanguage

Il est possible de coder ce comportement pour un scriptblock, on utilisera la méthode ***CheckRestrictedLanguage.*** On lui passe en paramètre les noms de commandes et de variables autorisées :

```
[string[]]$commandsAllowed = @( "ConvertFrom-StringData")
[string[]]$defaultAllowedVariables = @("PSCulture", "PSUICulture",
"true", "false", "null")
$allowEnvironmentVariables=$false
$Data={
  if ($PSVersionTable.PSVersion -eq ([Version]"2.0.0.0"))
  {"PowerShell 2.0"}
  else
  {"PowerShell >= 3.0 "}

  Get-ChildItem C:\
}
$data.CheckRestrictedLanguage($commandsAllowed,$defaultAllowedVariables,
$allowEnvironmentVariables)
```

Dans les deux cas les messages d'erreur sont identiques.

Si le premier paramètre est égal à *$null* toutes les commandes seront autorisées.

Si on passe *$null* dans le second paramètre, l'ensemble de variables par défaut sera autorisé. S'il s'agit d'une liste vide, aucune variable ne sera autorisée. Si c'est "*" alors n'importe quelle variable sera autorisée.

Le troisième paramètre autorise ou pas l'usage de variable d'environnement :

```
$Data={
  "var= $Env:PSModulePath"
}
$data.CheckRestrictedLanguage($commandsAllowed,$defaultAllowedVariables,
$true)
```

Pour connaitre les types autorisés on peut consulter le code source :

« Only allow simple types of arrays of simple types as defined by System.Typecode

***The permitted types are**: Empty, Object, DBNull, Boolean, Char, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime, String*

***We reject anything with a typecode or element typecode of object**...*

If we couldn't resolve the type, then it's definitely an error. »

2.2 Manifeste de module

Un manifeste de module est également exécuté dans le mode de langage restreint.

Si on déclare la clé *ModuleVersion* ainsi :

```
ModuleVersion=[Version]"1.0.0.0"
```

Comme précédemment ce code déclenche une exception lors l'import du module:

```
Import-Module Test
…
+ ModuleVersion=[Version]"1.0.0.0"
                 ~~~~~~~~~~
The type version is not allowed in restricted language mode or a Data
section.
```

Vous trouverez d'autres informations dans le document nommé Appendixes.

Note : le module *PowerShellGet* utilise une méthode plus restrictive.

3 Runspace contraint

Contraindre un runspace limite l'accès aux éléments du langage dans votre code et à la différence de la section *Data* c'est à vous de paramétrer le runspace.

On peut par exemple contraindre le langage dans le runspace de la session courante :

```
$host.Runspace.SessionStateProxy.LanguageMode = 'RestrictedLanguage'
```

Dès lors la lecture de cette même propriété devient impossible :

```
$host.Runspace.SessionStateProxy.LanguageMode
Property references are not allowed in restricted language mode or a
Data section.
```

A noter que l'usage d'Applocker bascule Powershell version 5 en mode contraint.

3.1 Créer l'état de session

Avant d'utiliser un espace de travail (*runspace*) on doit le configurer. On utilise pour cela un objet état de session (*SessionState*), cet état référence les objets accessibles pendant la durée de vie du runspace. Ces objets sont par exemple les variables, les fonctions, les fournisseurs, etc.

L'état de session est lié, entre autre, à la notion de portée, par exemple entre un module est une session console.

Créons un état de session et ajoutons lui la commande **Get-ChildItem** :

```
$InitialSessionState=
[System.Management.Automation.Runspaces.InitialSessionState]::Create()


$SessionStateCmdletEntry = New-Object
System.Management.Automation.Runspaces.SessionStateCmdletEntry 'Get-
ChildItem',([Microsoft.PowerShell.Commands.GetChildItemCommand]),$null


$InitialSessionState.Commands.Add($SessionStateCmdletEntry)
```

Ainsi seule l'exécution de cette commande est possible.

Pour retrouver le type du cmdlet on utilise **Get-Command** :

```
(Get-Command -Name 'Get-ChildItem).ImplementingType.Fullname
Microsoft.PowerShell.Commands.GetChildItemCommand
```

3.2 Créer le runspace

Voir le fichier de démo RS1.ps1

Ensuite on associe cet état de session configuré à un runspace. En lieu et place d'un objet runspace on utilisera la classe ***[Powershell]*** qui l'encapsule et facilite son usage. Ce qui implique d'associer à notre session Powershell le runspace configuré :

```
try {
 $Runspace= [RunspaceFactory]::CreateRunspace($InitialSessionState)
 $Runspace.Open()
 try {
  $PS = [PowerShell]::Create()
  $PS.Runspace = $Runspace
  $null=$PS.AddCommand("Get-ChildItem").AddParameter('Path','c:\temp\')
  $Results = $PS.Invoke()
  if ($PS.Streams.Error.Count -gt 0)
  {
    Write-Warning "Erreur"
    $PS.Streams.Error
  }
 }
 finally {
  $PS.Dispose()
 }
}
finally {
 $Runspace.Dispose()
}
```

Notez que l'on prend soin de disposer les objets créés et que l'on gère les possibles erreurs d'exécution directement sur la variable *$PS*.

En l'état ce code déclenche l'erreur suivante :

```
WARNING: Erreur
Get-ChildItem : Cannot find drive. A drive with the name 'C' does not
exist.
+ CategoryInfo          : ObjectNotFound: (c:String) [Get-ChildItem],
DriveNotFoundException
 + FullyQualifiedErrorId :
DriveNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
```

On peut exécuter la commande, mais celle-ci ne dispose pas d'un accès aux données.

En vérifiant la configuration :

```
$InitialSessionState | select-Object *
Providers                       : {}
Commands                        : {Get-ChildItem}
…
```

On constate que la liste des providers est vide, là où la configuration par défaut en contient de nombreux :

```
([Runspace]::DefaultRunspace.InitialSessionState.Providers).Name
Registry
Alias
Environment
FileSystem
Function
Variable
```

On doit donc ajouter explicitement le provider ***FileSystem*** à notre configuration :

```
$Provider = New-Object
System.Management.Automation.Runspaces.SessionStateProviderEntry
'FileSystem',([Microsoft.PowerShell.Commands.FileSystemProvider]),$null
$InitialSessionState.Providers.Add($Provider)
```

Voir le fichier de démo RS1-1.ps1

Cette configuration supplémentaire règle ce problème et l'on récupère bien les données demandées :

```
$Results
    Directory: FileSystem::C:\temp
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
             09/07/2017     21:18           2512 data.txt
…
```

Le fichier de démo RS1-2.ps1 reprend cet exemple en modifiant la commande exécutée *Get-ChildItem* par *Get-Process* :

```
$PS.AddCommand("Get-Process") > $null
```

Son exécution déclenche une exception dans la session principale et pas dans le runspace contraint :

```
Exception calling "Invoke" with "0" argument(s):
"The term 'Get-Process' is not recognized as the name of a cmdlet…"
+   $Results = $PS.Invoke()
    + FullyQualifiedErrorId : CommandNotFoundException
```

Notez également que pour une commande interdite la section DATA affiche son propre message qui explicite la cause de l'erreur. C'est donc au concepteur/trice de préciser ici la cause réelle de l'erreur. Ceci dit, le choix d'origine pour ce message d'erreur est intentionnel, il s'agit de ne pas donner plus d'information que nécessaire sur la cause de l'erreur.

3.2.1 Libération du runspace

Le fait d'associer un runspace créé par nos soins implique de prendre en charge sa libération :

```
try {
 $InitialSessionState=
[System.Management.Automation.Runspaces.InitialSessionState]::Create()
 $Runspace= [RunspaceFactory]::CreateRunspace($InitialSessionState)
 $Runspace.Open()
 try {
  $PS = [PowerShell]::Create()
  $PS.Runspace = $Runspace
  Write-Warning "Libération automatique : $($ps.IsRunspaceOwner)"
 }
 finally { $PS.Dispose() }
}
finally {  $Runspace.Dispose() }
WARNING: Libération automatique : False
```

Si on précise une création interne de runspace sa libération se fera automatiquement :

```
try {
  #Le runspace créé n'est pas contraint
 $PS = [PowerShell]::Create('NewRunspace')
 Write-Warning "Libération automatique : $($ps.IsRunspaceOwner)"
  $ps.Runspace>$null
 Write-Warning "Libération automatique : $($ps.IsRunspaceOwner)"
}
finally { $PS.Dispose() }
WARNING: Libération automatique : False
WARNING: Libération automatique : True
```

Notez que l'accès à la propriété Runspace modifie le contenu de '*IsRunspaceOwner*'**,** ce code n'appelant pas de méthode utilisant le runspace, cette propriété n'est pas mise à jour.

C'est un comportement particulier, car ici bien que l'on précise que l'instance de la classe ***[PowerShell]*** crée le runspace, la mise à jour de la propriété est retardée.

Pour un usage courant le problème ne se pose pas car l'accès, en interne, au runspace met à jour propriété '*IsRunspaceOwner*' :

```
try {
  $PS = [PowerShell]::Create('NewRunspace')
   $PS.AddCommand("Get-Process") > $null
   $Results = $PS.Invoke()
  Write-Warning "Libération automatique : $($ps.IsRunspaceOwner)"
}
finally { $PS.Dispose()}
WARNING: Libération automatique : True
```

3.3 Exécution de lignes de code

Le fichier de démo RS2.ps1 modifie de nouveau le code, cette fois on ajoute un script fonctionnellement identique :

```
$null=$PS.AddScript("Get-ChildItem -Path 'C:\Temp'")
```

Son exécution déclenche une exception :

```
Exception calling "Invoke" with "0" argument(s): "The syntax is not
supported by this runspace. This can occur if the runspace is in no-
language mode."
```

Vérifions notre configuration d'état de session :

```
$InitialSessionState | select-Object *
LanguageMode                    : NoLanguage
…
```

La valeur ***NoLanguage*** indique que l'on ne peut pas utiliser d'éléments du langage. Seules les commandes déclarées sont autorisées.

3.4 Modifier le mode de langage

Pour autoriser cette écriture le fichier de démo RS2-1.ps1 modifie le mode de langage du runspace :

```
$Mode= [System.Management.Automation.PSLanguageMode]::RestrictedLanguage
$InitialSessionState.LanguageMode=$Mode
```

Le mode *RestrictedLanguage* est celui utilisé par la section DATA, celui-ci offre quelques possibilités supplémentaire par rapport au mode *NoLanguage*, mais reste très limité.

Note :

Dans cet exemple la modification du mode en *FullLanguage* ne modifie pas la configuration des commandes, provider, etc.

Une session par défaut peut modifier le mode par défaut :

```
[Runspace]::DefaultRunspace.LanguageMode='ConstrainedLanguage'
```

Mais il n'y a pas de retour arrière possible :

```
[Runspace]::DefaultRunspace.LanguageMode='FullLanguage'
Cannot set property. Property setting is supported only on core types in
this language mode.
```

Pour consulter le détail des '*core types*', c'est-à-dire les types de base dont l'usage est considérés comme sans danger dans un runspace contraint.

Il est possible de modifier ce mode par GPO via la variable d'environnement nommée '***__PSLockdownPolicy***'.

3.5 Ajouter une variable

Si on souhaite injecter une variable existante dans le runspace on procède ainsi:

```
$Tab=@('Serveur1','Serveur2','Serveur3')
$Variable=Get-Variable -Name Tab


$VariableEntry = New-Object
System.Management.Automation.Runspaces.SessionStateVariableEntry`
$Variable.Name, $Variable.Value,$Variable.Description,'None'


$InitialSessionState.Variables.Add($VariableEntry)
```

'None' indique que la variable n'a pas de contrainte comme *AllScope* ou *ReadOnly*.

3.6 Importer une fonction

De même que pour une fonction :

```
Function Test { Write-output 'Test' }
 #Accès direct au ScriptBlock de la fonction
$Definition=${function:Test}
$SessionStateFunction = New-Object
System.Management.Automation.Runspaces.SessionStateFunctionEntry -
ArgumentList 'Test', $Definition
$InitialSessionState.Commands.Add($SessionStateFunction)
```

Si la fonction dépend d'autres ressources Powershell celle-ci doivent être explicitement déclarées dans le runspace via l'état de session.

3.7 Charger un module

Les modules sont chargés via la méthode *ImportPSModule()* d'une instance de la classe InitialSessionState.

On peut soit les charger par une spécification de module :

```
@{ModuleName="MyModule"; ModuleVersion="1.0.0.0"}
```

Soit par leur nom ou un nom complet d'accès.

La valeur par défaut de la variable *$PSModuleAutoLoadingPreference* est égale à ***$null***, dans ce cas une méthode interne renvoi la valeur '*All*'. Mais le module doit être explicitement chargé via l'état de session, sinon une exception vous l'indiquera clairement :

```
System.Management.Automation.CommandNotFoundException:
The 'Get-ScriptAnalyzerRule' command was found in the module
'PSScriptAnalyzer', but the module could not be loaded.
For moreinformation, run 'Import-Module PSScriptAnalyzer'.
```

4 Conclusion

Sachez qu'un runspace contraint n'est pas en soi une sécurité mais une brique de base permettant de concevoir un mécanisme de délégation (*constrained endpoint*) ou de s'intégrer à d'autres mécanismes de Windows comme indiqué dans cet article de blog.

Vous trouverez un exemple d'implémentation dans le module Plaster (création de template). Les templates consommés par ce module pouvant être importés depuis la galerie Powershell l'usage d'un runspace contraint limite l'accès à quelques fonctionnalités.