Module 7: Interaction avec la base de registre

Nous verrons maintenant comment on peut interagir avec la base de registres de Windows à partir de PowerShell. Nous verrons que ce n'est pas bien compliqué grâce à la magie des lecteurs PowerShell, qui permettront de traiter les clés de registres comme des répertoires.

Les lecteurs PowerShell

Dans PowerShell, certains éléments logiques du système d'exploitation peuvent être représentés sous forme de disques et utilisés comme tels. Vous avez déjà vu le disque "variable:" (le deux points est important) - faites cd variable: puis dir et vous verrez toutes les variables définies en ce moment dans votre session (équivalent à faire get-variable).

Il existe également un disque HKCU: (Hkey_Current_User) et un disque HKLM: (HKey_Local_Machine) qui permettent d'accéder aux deux ruches principales de Windows.

Pour voir tous les disques PowerShell définis sur votre système, faites simplement: get-psdrive.

Il est possible de créer des disques PowerShell qui pointent un peu partout - sur des clés (ou ruches) du registre, sur un répertoire (local ou réseau) ou sur certains autres aspects du système. Par exemple, on pourrait être intéressé à avoir un disque qui pointe vers HKey_Classes_Root, une autre ruche du registre. Pour ce faire, il suffit d'utiliser new-psdrive ainsi:

```
new-psdrive -name "HKCR" -psProvider "Registry" -root "Hkey_Classes_Root"
```

Et le tour est joué! Vous pouvez maintenant faire cd HKCR: pour atteindre ce disque.

Vous aurez deviné que le paramètre -name permet simplement de donner le nom que l'on veut à notre disque. -root permet de définir l'emplacement qui sera la racine du disque. Il reste -psProvider. Ce paramètre sert à définir le fournisseur du système qui se trouvera sur notre disque, ce qui est en quelque sorte le type de disque qui sera créé. Vous pouvez voir tous les fournisseurs existant en faisant get-psprovider. Les plus courants sont "Registry" et "FileSystem".

Les disques créés par l'usager ne sont pas permanents (ils devront être ajouté au profil si on veut les garder pour toujours).

Notez que vous pouvez détruire un disque en faisant remove-psdrive suivi du nom du disque.

Et une fois dans le lecteur?

Vous pouvez vous déplacer dans les ruches en faisant simplement cd HKCU: (par exemple). De là, les commandes dir et cd fonctionnent comme si les clés étaient des répertoires sur un disque. (N'oubliez pas, question de culture personnelle, que cd est un simple alias sur set-location et dir sur get-childitem.) Notez que vous pouvez appuyer sur Tab pour faire de l'autocomplétion, ce qui est bien pratique.

Notez toutefois que si les clés fonctionnent comme des répertoires, les valeurs ne fonctionnent pas comme des fichiers. En effet, les clés de registres ne sont là que pour permettre de classer les valeurs. Ce que l'on veut vraiment atteindre lorsque l'on se balade dans le regsitre, ce sont les valeurs.

Pourtant, si on fait ceci:

```
set-location HKCU:\Software\Microsoft\Windows\CurrentVersion\Run
```

on se retrouve dans la clé Run, qui contient normalement les chemins des logiciels qui démarrent avec Windows. Et un dir ne montre absolument rien. Essayons de remonter d'un niveau:

```
cd ..
dir
```

On voit bien Run qui semble contenir quelque chose. L'explication? Les valeurs ne sont pas des items, comme les clés, mais sont des propriétés des items représentant les clés. Du coup, pour les voir (et voir les données qu'elles contiennent), on doit faire par exemple:

```
get-itemproperty run
```

(on peut mettre n'importe quel chemin relatif ou absolu). On obtient alors quelque chose comme:

```
PSPath                  :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows
\CurrentVer sion\Run
PSParentPath            :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows
\CurrentVersion
PSChildName             : Run
PSDrive                 : HKCU
PSProvider              : Microsoft.PowerShell.Core\Registry
MultiG                  : "C:\Program Files\MultiG\MultiG.exe"
SetDefaultMIDI          : MIDIDef.exe
Google Update           :
"C:\Users\Regis\AppData\Local\Google\Update\GoogleUpdate.exe" /c
Gestionnaire Antidote.exe : C:\Program Files\Druide\Antidote\Gestionnaire
Antidote.exe
iTeleportConnect        : "C:\Program Files\iTeleport\iTeleport
Connect\iTeleportConnect.exe" -autostart
```

Les 5 premières lignes indiquent respectivement: le chemin de la clé, le chemin du parent de la clé, le nom de la clé (l'enfant), le lecteur PowerShell d'où elle provient et le "provider" qui permet au lecteur PowerShell de fonctionner.

Ce qui suit, ce sont les valeurs contenues dans la clé, dans le format "valeur : donnée".

Il existe une autre possibilité pour accéder aux valeurs: on peut utiliser une méthode de l'objet clé pour obtenir une liste de toutes ses valeurs ou pour obtenir la donnée dans une valeur précise.

En effet, chaque objet contient des propriétés (on en a utilisé en masse depuis le début), mais il contient aussi des méthodes, c'est-à-dire des fonctions qui peuvent être appelées à partir de l'objet pour travailler avec ou sur ses propriétés.

Par exemple, on peut utiliser dir (ou get-childitem) pour obtenir une clé précise et la stocker dans une variable:

```
$clés = get-childitem HKCU:\Software\Microsoft\Windows\CurrentVersion
```

Maintenant, $clés contient une collection de clés, les enfants de CurrentVersion. On peut s'en convaincre en affichant le contenu de $clés.

Comme pour toute collection, on peut aller chercher une case précise. Dans mon cas, la clé Run est

dans la case 16:

```
$clés[16]
```

Évidemment, dans un script, on pourrait parcourir tous les objets clés pour trouver celui qui fait notre affaire...

D'abord:

```
$clés[16] | get-member
```

nous informera des différentes propriétés utilisables. On peut en tester quelques-unes si on n'est pas certain de leur contenu. On se rend compte assez vite que PSChildName contient le nom "simple" de la clé et que Name contient le nom complet de la clé, avec le chemin. On peut donc faire:

```
$clés = get-childitem HKCU:\Software\Microsoft\Windows\CurrentVersion
foreach ($clé in $clés)
{
   if ($clé.PSChildName -eq "Run")

   {
      # On imagine un traitement ici...
   }
}
```

Pour appeler une méthode, on indique le nom de l'objet à partir duquel appeler la méthode, suivi d'un point, suivi du nom de la méthode (comme pour une propriété), **suivie des parenthèses ouverte et fermée**. Les parenthèses sont là pour un éventuel passage de paramètres, mais elles sont obligatoires même s'il n'y a pas de paramètres entre les deux.

Ici, on appellera GetValueNames() qui nous retournera un tableau de noms de valeurs (sous forme de strings):

```
$clés[16].getvaluenames()
```

Encore une fois, on pourrait mettre ça dans un tableau et les parcourir avec un foreach si nécsesaire...

On peut aller chercher la donnée à l'intérieur d'une valeur avec la méthode GetValue(). GetValue() prend en paramètre le nom de la valeur désirée (qui pourrait être contenue dans une variable...).

```
$clés[16].getvalue("MultiG")
```

Créer des clés et des valeurs

Évidemment, il y a des applets prévus pour ces tâches. new-item permet de créer un nouvel item sur un lecteur (ça peut donc être une clé de registre, un répertoire ou un fichier).

```
new-item -type registry test
```

Ceci créera une clé appelée test dans la clé courante. On peut utiliser un chemin relatif ou absolu avec le nom. (Notez que le type pourrait également

être "directory" ou "file" selon ce que l'on veut créer.)

Une fois la clé créée, on peut y ajouter des valeurs. Rappelons que ces valeurs sont en fait des propriétés de la clé. On utilisera donc l'applet set-itemproperty:

```
set-itemproperty -path "test" -name "valeur" -value "donnée"
```

Ceci ajoutera la valeur "valeur" contenant la donnée "donnée" dans la clé "test". Notez que l'on peut donner un chemin complet vers la clé (absolu ou relatif) et que le nom de la valeur peut être "(défaut)" pour la valeur par défaut de la clé.

La création d'une clé à partir des méthodes est pas mal plus complexe que sa lecture... En effet, par défaut, les clés contenues dans des objets sont considérées comme *read-only* (accessible en lecture seulement) par les méthodes. Il faut donc faire une passe-passe pour ouvrir la clé en mode *read-write*, ce qui demande d'utiliser une méthode statique (et la syntaxe peut paraître étrange). Je vous donne le code purement pour votre culture personnelle ou pour ceux d'entre vous qui ont envie d'explorer plus loin.

```
$reg =
[Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey("CurrentUser
", env:ComputerName)
$regKey= $reg.OpenSubKey($clé, $true)
$regkey.SetValue("valeur","donnée","String")
```

Là-dedans, on comprend que env:ComputerName retourne le nom de l'ordinateur (via une variable d'environnement Windows - voyez comme c'est joli!) et que $clé doit contenir un chemin complet vers une clé de HKCU. On peut remplacer "CurrentUser" par "LocalMachine" ou "ClassesRoot", et "String" par "Dword", "Binary" ou tout autre type de valeur de registre. Le $true est là pour indiquer le mode *read-write* et est donc essentiel.

Notez que la première ligne représente un appel à une méthode statique! Sympathique, n'est-ce pas?

Un exercice

Votre script devra parcourir une arborescence de clés à la recherche de **clés** et de **valeurs** (**pas de données**)qui contiennent un mot spécifié par l'usager.

Il devra d'abord demander un chemin de départ (qui devra être un chemin vers une clé de registre - pas besoin de valider ça, on suppose que c'est le cas). Il devra toutefois vérifier que c'est une clé qui existe, sinon, il devra en redemander un à l'usager. (Il se trouve que test-path fonctionne aussi bien avec des clés de registres qu'avec des

répertoires, grâce à la magie des lecteurs PowerShell, alors je vous réfère à la procédure du labo 2 qui devrait maintenant vous paraître simple).

Ensuite, il demandera un mot-clé à trouver.

Puis la recherche commence. Le script doit:

- Parcourir toutes les clés à partir du point de départ. Encore une fois, ça ressemble à ce que vous avez fait pour le labo 2: un get-childitem dans un foreach devrait être fort pertinent. Assurez-vous d'utiliser le paramètre -recurse de get-childitem qui fera en sorte de parcourir récursivement toute l'arborescence (et non pas juste la clé de départ).

    - Pour chacune des clés ainsi traversées, le script devra tester si le nom de la clé contient le mot donné par l'usager (qui peut être n'importe où dans le nom, je vous laisse trouver comment faire ça). Si le nom complet de la clé contient le mot, votre script devra afficher le nom complet de cette clé.

    - Toujours pour chacune des clés traversées, votre script doit récupérer les valeurs qui s'y trouvent et les vérifier une à une pour voir si elles aussi ne contiendraient pas le mot-clé de l'usager. En effet, ce mot peut autant se trouver dans le nom d'une clé que dans celui d'une valeur! (Ça sent la boucle imbriquée!) Les valeurs qui contiennent le mot-clé devront aussi être affichées, mais un peu en retrait et précédé d'une flèche (->) pour montrer que ce sont des valeurs comprises dans la clé précédente.

Par exemple, si l'usager spécifie comme point de départ "HKCU:\" (carrément!) et comme mot-clé "Windows", la sortie devrait ressembler à ceci:

```
HKEY_CURRENT_USER\AppEvents\Schemes\Apps\.Default\WindowsUAC\.Default
HKEY_CURRENT_USER\AppEvents\Schemes\Apps\.Default\WindowsUAC\.Modified
  -> WindowSize
  -> WindowsEffect
  -> DragFullWindows
HKEY_CURRENT_USER\Software\Adobe\Audition\3.0\PluginCache\en_US\ImporterWindowsM
edia.prm
HKEY_CURRENT_USER\Software\Adobe\Audition\3.0\PluginCache\en_US\ImporterWindowsM
edia.prm\Importer
  -> Windows Character Set
```

Évidemment, la sortie sera **beaucoup** plus longue que ça, je ne vous donne qu'un échantillon pertinent.

Il est possible que votre script ne puisse pas entrer dans certaines clés et dans ce cas, des messages d'erreur seront constamment affichés (notez que le script continuera tout de même son exécution). Ces messages pollueront la sortie inutilement, alors voici un petit truc pour s'en débarrasser:

```
$ErrorActionPreference="SilentlyContinue"
```

En effet, la variable $ErrorActionPreference contient l'instruction à suivre en cas d'erreur. On peut la définir quand on veut (au début d'un script, par exemple). "SilentlyContinue" signifie de ne pas afficher de message et de continuer le script comme si de rien n'était - pas très prudent ni très intelligent, mais dans le cas d'un script simple comme celui-ci, c'est sans danger.

Notez en passant que les valeurs possibles pour $ErrorActionPreference sont SilentlyContinue, Continue (c'est le mode par défaut), Stop et Inquire (demander quoi faire à l'usager). Notez aussi que cette variable s'applique uniquement pour les erreurs sans fin d'exécution (les erreurs avec fin d'exécution mettent toujours fin à l'exécution et on ne peut rien y faire).