PowerShell

- Quick commands (Windows)
- Check if script runs as admin
- Scripting Guidelines
- Authenticate to different host
- Graph API Handling via PowerShell
- Restructure JSON (object) into hashtable
- Perform user interactions in the system context
- Use of simultaneous processing using PowerShell threads

Quick commands (Windows)

Nameresolution via other server than default

```
nslookup somewhere.com some.dns.server
```

Get Windows client join status

```
dsregcmd /status
```

Get applied GPO status on Windows client

```
gpresult /h C:\LocalData\GPResult.html
```

Get RDS license status

```
(Invoke-WmiMethod -PATH (gwmi -namespace root\cimv2\terminalservices -class win32_terminalservicesetting).__PATH -name GetGracePeriodDays).daysleft
```

```
wmic /namespace:\\root\CIMV2\TerminalServices PATH Win32_TerminalServiceSetting WHERE (__CLASS !="") CALL GetGracePeriodDays
```

Rejoin device to Microsoft Entra ID

```
dsregcmd /forcerecovery
```

Restart Computer to BIOS/UEFI

```
shutdown /r /fw
```

Delete specific files recursive

```
$source = "<sourcepath>"
$fileextension = "<filextension>"
get-childitem $source -include *$fileextension | foreach ($_) {remove-item $_.fullname}
```

Run PowerShell as admin from different session

```
Start-Process PowerShell -verb runas
```

Get last reboot time

```
Get-WmiObject win32_operatingsystem | select csname,
@{LABEL='LastBootUpTime';EXPRESSION={$_.ConverttoDateTime($_.lastbootuptime)}}
```

Delete Windows Hello Container

```
certutil.exe -DeleteHelloContainer
```

Reset Windows to Factory Settings

```
systemreset
```

Set error action in PowerShell session

```
$erroractionpreference = "silentlycontinue"
```

Test network connection

Test TCP port

```
Test-NetConnection -ComputerName <youripaddress> -Port <yourport>
```

```
$ipaddress = "<youripaddress>"
$port = "<yourport>"
$connection = New-Object System.Net.Sockets.TcpClient($ipaddress, $port)
if ($connection.Connected) {
    Write-Host "Success"
} else {
    Write-Host "Failed"
}
```

Test UDP port

```
$ipaddress = "<youripaddress>"
$port = "<yourport>"
$connection = New-Object System.Net.Sockets.UdpClient($ipaddress, $port)
if ($connection.Connected) {
    Write-Host "Success"
} else {
    Write-Host "Failed"
}
```

Check if address goes through proxy

```
([System.Net.WebRequest]::GetSystemWebProxy()).getProxy("https://lucanoahcaprez.ch")
```

This returns an object containing the information about the connection. If the "Host" property does not contain the original requested URL, there is a setting on Windows present. This setting defines the Proxy and the address of the proxy can be viewed in the "Host" property on the response object of this command.

Manage Local Windows Groups

Add Microsoft Entra ID user to administrator group

```
$Email = ""
Add-LocalGroupMember -SID "S-1-5-32-544" -Member "AzureAD\$EMail"
```

Add Microsoft Entra ID user to remote desktop group

```
$Email = ""
Add-LocalGroupMember -SID "S-1-5-32-555" -Member "AzureAD\$EMail"
```

Open Windows Control Panel pages

Printer overview

```
explorer shell:PrintersFolder
```

Turn On/Off Windows Feature

```
optionalfeatures
```

Advanced System Properties

```
SystemPropertiesAdvanced
```

PowerShell Module Management

List installed modules

```
Get-InstalledModule
```

Manage Windows File System

Add System Hardlink

```
mklink /d <C:\MyLocalFolder> <X:\MyRemoteFolder>
```

Check if script runs as admin

Check session for admin privileges

If you have to run your commands in your custom script as admin, it is essential for error handling to check if the current session is admin.

```
$isAdmin = (New-
Object Security.Principal.WindowsPrincipal $CurrentUser).IsInRole([Security.Principal.SecurityIdentifier] "S-1-5-
32-544")

if($isAdmin){
Write-Output "PowerShell session is admin"
}
if(!$isAdmin){
Write-Output "PowerShell session is not admin"
}
```

Use case

This script can be used to combine with a Remote PowerShell Session to get local Configurations from other Windows Servers or Clients, which only can be accessed via local admin privileges or domain admin privileges.

More about this use case in docs entry "Authenticate to different host".

Scripting Guidelines

Variable naming and notation

Variables are named in the PascalCase notation. Accordingly, the main activity of the script is taken and the words are concatenated. In the description, the first letter of each word must be capitalized:

I am doing something → IAmDoingSomething

The naming of the script should also be done in English.

Comments

Comments are important in PowerShell to recognize the context of the code and function bice. Comments are started with "#" and then not executed in the code. Here it is important not to describe what is done in detail, but why and what dependencies need to be considered. PowerShell is an easy to read language and can therefore be reverse engineered very quickly, what the code really does, accordingly explanations are not necessary and only lengthen the code unnecessarily.

Authenticate to different host

Use case

This script block can be used in combination with an Azure Runbook. For example you can run a PowerShell script on an Active Directory Domain Controller via an AD Joined Hybrid Worker. So, you can use all the advantages of Azure Runbooks with the ability to automate the core of Active Directory. In addition, an external source can dynamically check all AD DCs and scheduled tasks do not have to be manually installed on all domain controllers for the same use case.

PowerShell Example

This code snippet can be used to authenticate to a host (Server) and use different credentials for the connection. This script is specific to check if the user account in $ServiceAccountUPN has local admin access on the host in $ServerName. To customize the code which will be executed on the remote machine, you have to change the code inside the -ScriptBLock {<insertcustomcodehere>}.

```
$ServiceAccountUPN = ""
$ServiceAccountPW = ""
$ServerName = ""

$Password = ConvertTo-SecureString -AsPlainText $ServiceAccountPW -Force
$Credential = New-Object System.Management.Automation.PSCredential($ServiceAccountUPN, $Password)

$output = Invoke-Command -Credential $Credential -ComputerName "$ServerName" -ScriptBlock {
   $CurrentUser = [Security.Principal.WindowsIdentity]::GetCurrent()
   $isAdmin  = (New-
Object Security.Principal.WindowsPrincipal $CurrentUser).IsInRole([Security.Principal.SecurityIdentifier] "S-1-5-
32-544")
   write-output "Output $($CurrentUser) ($($isAdmin))"
}

$output
```

Invoke-Command uses Windows Remote Management under the hood.

Windows Remote Management

Windows Remote Management (WinRM) uses the Port: **5986** over TCP. In the background is HTTPS Protocol. WinRM is automatically installed with all currently supported versions of the Windows operating system. The WinRM service starts automatically on Windows. By default, Internet Connection Firewall (ICF) blocks access to ports.

Graph API Handling via PowerShell

> Requirements: An App Registration with the appropriate permissions and a ClientSecret.

Graph API Authentication

First, the authentication header must be compiled in the script. With this header (here the variable $Header) the authentication at the Graph API can be executed. The top three variables now contain the values, which were compiled in an upper point.

```
$TenantID = "<tenantid>"
$ClientId = "<cliendid>"
$ClientSecret = "<clientsecret>"

$Body = @{
"tenant" = $TenantId
"client_id" = $ClientId
"scope" = "https://graph.microsoft.com/.default"
"client_secret" = $ClientSecret
"grant_type" = "client_credentials"
}

$Params = @{
"Uri" = "https://login.microsoftonline.com/$TenantId/oauth2/v2.0/token"
"Method" = "Post"
"Body" = $Body
"ContentType" = "application/x-www-form-urlencoded"
}
$AuthResponse = Invoke-RestMethod @Params

$Header = @{
   "Authorization" = "Bearer $($AuthResponse.access_token)"
```

```
}
```

Graph API Resources - Getting Information

This is a simple example query to get information. This only reads out. By the method "GET" this can be recognized on the second line.

```
$Email = "<youremailadress>"
$User = Invoke-RestMethod -Method GET -Uri "https://graph.microsoft.com/v1.0/users/$Email" -ContentType "Application/Json" -Header $Header
```

The following is the output from the $User variable, which has been populated in the top line with information from the Graph API.

```
@odata.context    : https://graph.microsoft.com/v1.0/$metadata#users/$entity
businessPhones    : <yourbusinessphones>
displayName       : <yourdisplayname>
givenName         : <yourforename>
jobTitle          : <yourjobtitle>
mail              : <youremailadress>
mobilePhone       : <yourmobilephonenumber>
officeLocation    : <yourofficelocation>
preferredLanguage : <yourpreferredlanguage>
surname           : <yoursurname>
userPrincipalName : <yourupn>
id                : <youruserid>
```

Graph API Resources - Create information

In the following example, an entity is created via the Graph API in Intune. Here, the necessary information is now also transmitted, using JSON Body.

```
$KGTag = "TST"
$ScopeTagProdName = "SCT-INT-$KGTag-INTUNE-KGObjects-PROD"
$ScopeTagProdBody = @"
```

```
{
   "displayName":"$ScopeTagProdName",
   "description":"ScopeTag for Company $KGTag"
}
"@
$global:ScopeTagProd = Invoke-RestMethod -Method POST -Uri
"https://graph.microsoft.com/beta/deviceManagement/roleScopeTags" -ContentType "Application/Json" -Header
$Header -body $ScopeTagProdBody
```

$global:ScopeTagProd is a global variable which has been populated with the return of the graph query above. The content of the variable is as follows:

```
id displayName □□□□□□  description               isBuiltIn
-- ----------- □□□□□□  -----------            □□--------
45 SCT-INT-TST-INTUNE-KGObjects-PROD   ScopeTag for Company TST     False
```

Restructure JSON (object) into hashtable

This script is needed if you get an object by any source (e.g., json) and you have to give every member of the object as a value (keypair) into a hash table.

```
$bodyjson = @"
{
   "logtype": "testlogs2",
   "logbody": {
      "computername":"Device-123456",
      "user":"luca",
      "ouput":"registry key was set",
      "status":"success",
      "time":"xyz"
   }
}
"@

$bodyobject = ConvertFrom-Json $bodyjson
$logtype = $bodyobject.logtype
$logbodyobject = $bodyobject.logbody

$logbodyobjectmember = Get-Member -InputObject $logbodyobject | where {$_.Membertype -eq
"NoteProperty"}

$outputproperties = @{}

foreach ($item in $logbodyobjectmember.name){
   $outputproperties.add($item, $logbodyobject.$item)
}
```

Use case

To post data into a log analytics workspace you have to send a hash table as body of the post request. If you build an API via Azure Functions then you get a JSON object as input. so you have to restructure the incoming body to be a hash table. This has to be dynamic so the input length and member entities can change.

Perform user interactions in the system context

Usage of ServiceUI

ServiceUIx64.exe can be used to execute a certain part of a script in the user context, even though you execute the main script in the system context.

```
ServiceUI.exe -process:TSProgressUI.exe %windir%\sysnative\WindowsPowerShell\v1.0\powershell.exe -WindowStyle Hidden -NoProfile -ExecutionPolicy Bypass -nologo -File <scriptpath>
```

Link collection

Use ServiceUI With Intune To Bring SYSTEM Process To Interactive Mode HTMD Blog (anoopcnair.com)

How to display a custom window in SCCM Task Sequence using PowerShell | Slightly Overcomplicated

Use of simultaneous processing using PowerShell threads

If a long script has to perform many tasks, this can take a lot of time in PowerShell. As PowerShell is not multithreaded by nature and processes are always executed sequentially, so-called threads can be used to minimise the script length.

Example Code

This code serves as a template for the parallel execution of processes. The tricky part with threads is the handling of input parameters and output values.

The script part is defined in the $Scriptblock variable. There you can work with arguments that are passed to the thread. They can be used with the $args variable and the element number. Values can be returned using the keyword "return".

```
$Scriptblock = {
	Start-Sleep 5
	return $args[0]
}

$MaxThreads = 10
$RunspacePool = [runspacefactory]::CreateRunspacePool(1, $MaxThreads)
$RunspacePool.Open()

$Jobs = @()
$InputObjects = @(
	"<yourinputobject1>",
	"<yourinputobject2>",
	"<yourinputobject3>",
	"<yourinputobject4>",
	"<yourinputobject5>"
```

```
)
$OutputObjects = @()

foreach ($Object in $InputObjects) {
	$Instance = [powershell]::Create()
	$Instance.RunspacePool = $RunspacePool
	$Instance.AddScript($ScriptBlock).AddArgument($Object) | Out-Null
	$Jobs += New-Object PSObject -Property @{
		Instance = $Instance
		State    = $Instance.BeginInvoke()
	}

}
while ($Jobs.State.IsCompleted -contains $false) {
	Start-Sleep 1
}
foreach ($job in $jobs) {
	$OutputObjects += $job.Instance.EndInvoke($job.State)
}

$Instance.Dispose()
```