Lab 2: PowerShell Recap and Scripting

Windows Server Labs

<PLACEHOLDER NAME>

Academic Year: 2021-2022

Table of Contents

[Introduction 3](#_Toc62484178)

[Learning Goals 3](#_Toc62484179)

[Knowledge (what you need to know) 3](#_Toc62484180)

[Abilities (what you need to be able to do) 3](#_Toc62484181)

[Requirements 3](#_Toc62484182)

[Combining cmdlets in a script 4](#_Toc62484183)

[Standing on the shoulder of giants 5](#_Toc62484184)

[Let’s bring it all together 6](#_Toc62484185)

[Extensions - Optional Assignments 9](#_Toc62484186)

## Introduction

This lab should give a quick recap of PowerShell on the command line. Additionally we will bundle statements into a file and take a look at some basic scripting for future automation.

## Learning Goals

# Knowledge (what you need to know)

1. Recap PowerShell and its cmdlets.
2. Know the difference between PowerShell version 5.x and 7.x.
3. Know how to create a basic PowerShell script.
4. Know the basic data structures (array and hash table) in scripting and PowerShell.
5. Know how to read from file use parameters and write to stdout and file.

# Abilities (what you need to be able to do)

1. Integrate all of the above described under knowledge.
2. Make use of the built-in documentation and help.
3. Create a script that bundles different operations and performs a set of tasks.
4. Parameterize PowerShell scripts.

## Requirements

None

## Combining cmdlets in a script

PowerShell should (!) ring a bell as we have used it before. In the past we used the so-called **cmdlets** (**Verb-Noun** structured commands) to perform operations on the command line (“Windows PowerShell” or “Windows PowerShell (x86)” which you shouldn’t use anymore).

If you search on for PowerShell on Windows you might have noticed “**Windows PowerShell ISE**”. ISE stands for “**Integrated Scripting Environment**”. It is more or less some sort of an IDE for PowerShell scripting. Some sources ([How to replicate the ISE experience in Visual Studio Code - PowerShell | Microsoft Docs](https://docs.microsoft.com/en-us/powershell/scripting/dev-cross-plat/vscode/how-to-replicate-the-ise-experience-in-vscode?view=powershell-7.1)) claim that the ISE is deprecated and the proper way to script is using VS Code with proper extensions. We advise to explore the possibilities (GUI-tools) of the ISE but in the end you are free to write scripts the way you see fit! 😉

PowerShell has a lot of alias or shortcuts built-in in the language. In scripts however the convention is to be as verbose as possible. Therefore always use the full cmdlet as much as possible as it improves readability.

As of today there are typically **2 PowerShell version flavors**. There is the standard 5.x that comes with Windows 10/11 (and Servers) and there is PowerShell core (7) that is cross-platform (in other words it runs on Linux as well): [https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell](https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-7.1)

In this course we will focus mainly on the default 5.x version that ships with Windows but most exercises/cmdlets/scripts should work fine on PowerShell 7.x. In fact as an *optional assignment* you are challenged to test and verify your commands and scripts on PowerShell 7.x and document any curiosities. To find out **which version of PowerShell** you are running you can issue the following variable:

$PSVersionTable

1. If you look at the results of this variable. What type of Variable is this? (hint: Get-Member)

System.Collections.Hashtable

1. In the output there is a name called “PSCompatibleVersions” that holds more information than is displayed. Use the . notation to display the full contents.

$PSVersionTable.PSCompatibleVersions

1. Open the PowerShell ISE and copy paste the following lines of code. Try to run it in the ISE (without saving it to a file) and afterwards safe it to “lab2q3.ps1”. Note the ps1 extension. Finally try to run the script from a normal PowerShell window (not ISE) and verify that everything works the same. Note the foreach, note the $($username.Name) (try changing it to $username.Name instead and check the output). What is the type of $usernames? Is $username the same type?

$usernames is an array of Hashtables

$username is an Hashtable

Write-Host "Hello world, this is the start of the script" -ForegroundColor Green

$localusers = Get-LocalUser

$usernames = $localusers | Select-Object -Property Name

foreach($username in $usernames) {

    write-host "There exists a local user with the name: $($username.Name) on this machine."

}

Write-Host "Goodbye world, this is the end of the script" -ForegroundColor Yellow

1. Wouldn’t it be cool if we can colorize the name of the user in red but only the name? There is no real option to Write-Host that allows this. A quick search online results in a PowerShell Module called “PSWrite-Color”. Install this module and change the Write-Host in the foreach to Write-Color in such a way that the username becomes red.

Text

Description automatically generated

## Standing on the shoulder of giants

1. As a rule of thumb a lot of scripting is possible as soon as you can read in, write out, parse/iterate/enumerate. Take a look at the following example, run this script and have a look at the arguments of the script with ctrl+space on the CLI and take notes on how the following things are implemented and used in PowerShell:
   1. An array 🡺 $array = @(“one”, “two”, “three”)
   2. A hash table 🡺 $hash = @{ $key = $value }
   3. Parameters for the script 🡺 [Parameter(Mandatory)][int] $index
   4. $\_ 🡺 'THIS' token. It refers to the current item in the pipeline.
   5. @{Name=…; Expression = {} } 🡺 hashtable
   6. Take a look at the about page for functions and create an example sum(a,b).

param([Parameter(Mandatory)][int] $index, [int] $key, [string] $value, $outfile='outfile.txt')

$array = @("one", "two", "three")

Write-Host "Contents of the array:" -ForegroundColor Green

for ($i = 0; $i -lt $array.Count; $i++) {

    Write-Host "`t" $i $array[$i] -ForegroundColor Yellow

}

$array | ForEach-Object {

    Write-Host "`t" $\_ -ForegroundColor Red

}

Write-Host "Content of array on $index :"$array[$index] -ForegroundColor Green

$hash = @{

    $key = $value

}

Write-Host "Contents of the hash keys: $($hash.keys)" -ForegroundColor Green

Write-Host "Contents of the hash values: $($hash.values)" -ForegroundColor Green

Write-Host "Writing local users to $outfile..."

(Get-LocalUser | Select-Object Name, Description, @{Name="MyColumn - LastLogon"; Expression={$\_.LastLogon} } ) | Out-File $outfile

## Let’s bring it all together

1. In this exercise we will create a script that parses a file (*sshd.log* and *auth.log* (see Leho)). The goal of this script is to gain insight in the number of different IP addresses that connected to (or try to connect to) the SSH-server. Create a PowerShell script that combines all previous techniques. Below is a step-by-step guide for a **possible** solution. The format of sshd.log looks something like this:

2021-01-20T13:55:04 sshd[42591] Connection closed by 46.36.103.1 port 63833 [preauth]

2021-01-20T13:46:50 sshd[23319] Connection closed by 46.36.103.1 port 24843 [preauth]

2021-01-20T13:41:17 sshd[34874] Connection closed by 159.89.110.115 port 42002

2021-01-20T13:41:17 sshd[34874] **error:** kex\_exchange\_identification: Connection closed by remote host

2021-01-20T13:41:17 sshd[34874] **error:** kex\_exchange\_identification: Connection closed by remote host

2021-01-20T13:38:45 sshd[30499] Connection closed by 46.36.103.1 port 42352 [preauth]

2021-01-20T13:36:09 sshd[34794] Disconnected from 221.181.185.223 port 64419 [preauth]

2021-01-20T13:36:09 sshd[34794] Received disconnect from 221.181.185.223 port 64419:11:  [preauth]

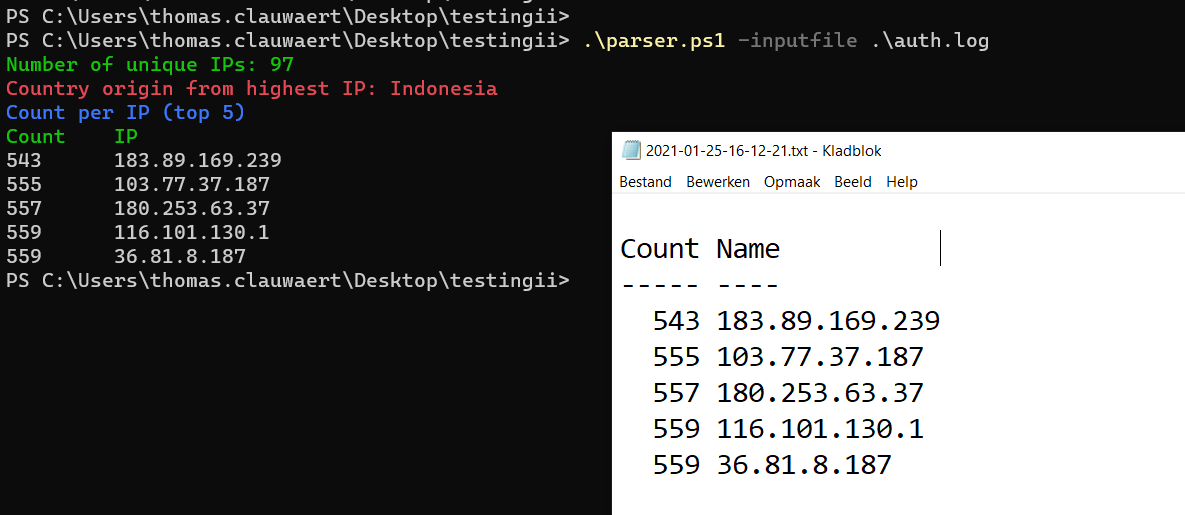
2021-01-20T13:30:39 sshd[55950] Connection closed by 46.36.103.1 port 59873 [preauth]

2021-01-20T13:28:50 sshd[54769] Disconnected from authenticating user root 5.252.224.197 port 50560 [preauth]

2021-01-20T13:28:50 sshd[54769] Received disconnect from 5.252.224.197 port 50560:11: Bye Bye [preauth]

2021-01-20T13:28:23 sshd[31285] Connection closed by invalid user pi 78.148.231.38 port 36148 [preauth]

The end-goal can be seen on the following screenshot:



Note the parameters, the name of the txt-file. You don’t have to worry about the first enter or the column-name “name” in the txt-file.

**Step-by-step Guide:**

* Start by setting up the parameters.
  + Create a parameter $topnr with the **default** value “5” of the type int
  + Create a **mandatory** parameter $inputfile
  + Create a parameter $outfile that will result the output of a specific date format (yyyy-MM-dd-HH-mm-ss)
* Create a variable $file with the contents of $inputfile.
* Create an empty array $ips
* Loop over every $line in the $file-variable
  + Create a $regex variable that checks if $line follows a specific pattern: IP-address (4 decimals separated by a .
  + For every match fill the $ips array with the result. Hint: Matches.Groups[index].Value
* Create a $sorted\_unique\_ips with the use of Sort-Object and Get-Unique
* This is the result of the first line: “Number of unique IPs: …”
* Create a $group object by piping the $ips array to Group-Object and explore the output.
* Find out the IP with the highest frequency in the file by using Sort-Object and Select-Object
* The combination of “Invoke-WebRequest” and “ifconfig.co/country?ip=$(…)” should help you for the country.
* Finally create an $output variable printing the $topnr number of lines and use $output | Out-File “$outfile.txt” to write to the txt-file.

There are other ways to solve this specific use case (some easier, some harder). As long as you get the same output and functionality (writing to the txt-file with the correct name) you succeeded.

**Post a screenshot and your script below** just like in the screenshot above but instead of auth.log use sshd.log. Make sure all information such as unique IPs, Country, top 5, txt-file contents and name are visible.

Text

Description automatically generated

## Extensions - Optional Assignments

For SE students:

* Try to perform the last scripting exercise (reading in, parsing, writing out) in your favorite programming language and search for the best practices
* Install PowerShell (core) 7 and test if everything works as intended.
* Expand the script and provide additional options such as “the number of Invalid users”, “the invalid users”, …

For CSP students:

* Try to perform the last scripting exercise in Python (3) as a lot of security-related scripts and PoC’s are written in Python.
* Install PowerShell (core) 7 and test if everything works as intended.
* Expand the script and provide additional options such as “the number of Invalid users”, “the invalid users”, …