Lab 1: PowerShell Basics and Gathering Host Information

Vijaysingh Puwar

Department of Computer Science, Pace University

CYB 631: Automating Information Security with Python and Shell Scripting

Professor Alex Tsekhansky

September 4, 2025

Contents

Exercise I: Starting with PowerShell and Simple Commands 6

Step 1 — Launch Windows PowerShell from the Run dialog 6

Step 2 — Verify the PowerShell console opened 6

Step 3 — Clear the console for a clean workspace 7

Step 4 — Create a New Folder on the Desktop 8

Step 5 — Verify the Logged-In User 9

Step 6 — Review Directory Contents and Confirm Current Path 9

Step 7 — Navigating with pushd and popd 10

Step 8 — Create a Dedicated Folder for Class Files 11

Step 9 — Create and Display a New Text File 12

Step 10 — Store File Path in a Variable and Access Contents 13

Step 11 — Modify File Attribute to Read-Only 14

Step 12 — Revert File Attribute and Verify Edit Access 15

Exercise II: Simple Object Operation 17

Step 1 — Concatenate Strings into a Single Object 17

Step 2 — Assign String to a Variable and Inspect Member 17

Step 3 — Inspect String Properties and Methods 18

Exercise I I I: Gather Process Information 20

Step 1 — List All Running Processes 20

Step 2 — Filter Processes Starting with “W” 22

Step 3 — Store Notepad Process in a Variable and Attempt to Kill It 22

Step 4 — Show Processes with ID ≥ 10,000 (Pipeline, Sorted Table) 23

Exercise IV: Useful PowerShell Commands 25

Step 1 — Explore Help and List Available Commands 25

Step 2 — Explore Members of the Get-Process Object 25

Exercise V: First PowerShell Script 27

Step 1 — List Processes Beginning with "N" 27

Step 2 — Sum Handles of "N" Processes 28

Step 3 — Run the script and manage the execution policy 28

Step 4 — Create and Run script1.ps1 30

Exercise VI: Develop Your Own PowerShell Script – CPU Time 32

Step 1 — Create a Script to Identify Top CPU-Consuming Processes 32

Exercise VII: Lab and Class Reflection 33

What I liked about this lab: 33

Challenges I encountered: 33

Suggestions for improving the class: 33

Turning off virtual machines: 33

Summary

This lab introduced me to the fundamentals of PowerShell for both basic command usage and more advanced scripting. Beginning with simple tasks such as launching PowerShell, confirming the logged-in user, and creating directories, I developed confidence in navigating the environment and organizing files effectively. As the lab progressed, I created and modified text files, experimented with file attributes, and practiced managing file access by toggling read-only permissions. These exercises reinforced the importance of precision, since even minor syntax errors produced immediate feedback, highlighting how accuracy is essential in scripting environments.

Moving into object operations, I learned how PowerShell treats all data as objects with associated properties and methods. By concatenating strings, assigning them to variables, and applying functions such as .Length and .ToLower(), I recognized how object-oriented features make data manipulation both powerful and intuitive. This understanding became particularly important when working with processes, where I listed active tasks, filtered them by name, stored them in variables, and even terminated specific instances such as Notepad. These exercises demonstrated how administrators can monitor and control system behavior directly through the shell, an essential capability for both security operations and troubleshooting.

The scripting portion of the lab emphasized automation and efficiency. By summing handle counts from processes and later writing cputime.ps1 to identify top CPU-consuming tasks, I experienced the value of turning repetitive commands into reusable scripts. Challenges arose when execution policies initially blocked my scripts, but I overcame this by learning how to apply a process-scoped bypass safely. This not only enabled me to run my own scripts but also deepened my understanding of how PowerShell balances functionality with security.

Overall, this lab was both practical and insightful. I appreciated the structured approach, which connected basic commands with meaningful administrative tasks and eventually to

scripting that automates system monitoring. The experience helped me bridge theory and practice while also highlighting areas where PowerShell directly supports cybersecurity, such as resource monitoring and process control. The reflections on execution policies and security considerations will be especially valuable as I continue to use scripting for automation in future labs and real-world scenarios.

Exercise I: Starting with PowerShell and Simple Commands

This exercise begins by launching Windows PowerShell, verifying the shell opened to the user profile directory, and clearing the console to prepare for subsequent commands.

Step 1 — Launch Windows PowerShell from the Run dialog

I opened the Windows **Run** dialog and typed **PowerShell**, then selected **OK** to start the shell. See Figure 1.

Commands/keys used: Win + R → type PowerShell → **OK**

Observed result: Windows PowerShell launched.

Notes: Using the Run dialog ensures I start a fresh session with default settings.

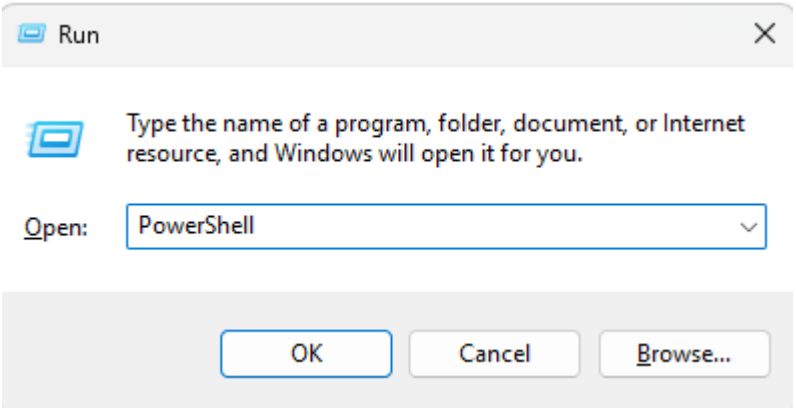

Figure 1: Opening Windows PowerShell via the Windows Run dialog

Step 2 — Verify the PowerShell console opened

A new PowerShell console window appeared with the default banner and the prompt set to my user context:

PS C:\Users\vp32780n> (see Figure 2).

Observed result: The shell opened in my home directory under C:\Users\vp32780n.

Why these matters: Confirming the starting directory and user context helps avoid path mistakes in later steps.

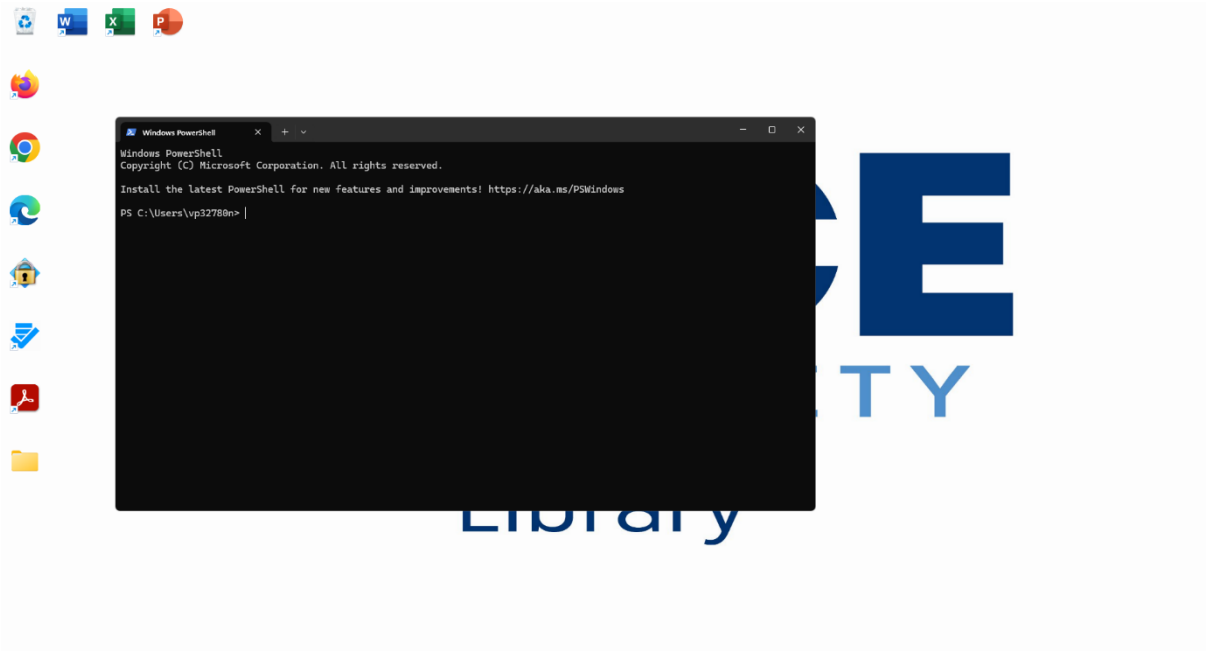

Figure 2: Fresh PowerShell session showing the default prompt in the user profile directory

Step 3 — Clear the console for a clean workspace

I ran the clear command to remove previous output from the screen (see Figure 3).

Command used: clear (alias of Clear-Host)

Observed result: The console cleared and returned to a blank prompt line PS C:\Users\vp32780n>.

Why this matters: Clearing the console improves readability for screenshots and separates output from different tasks.

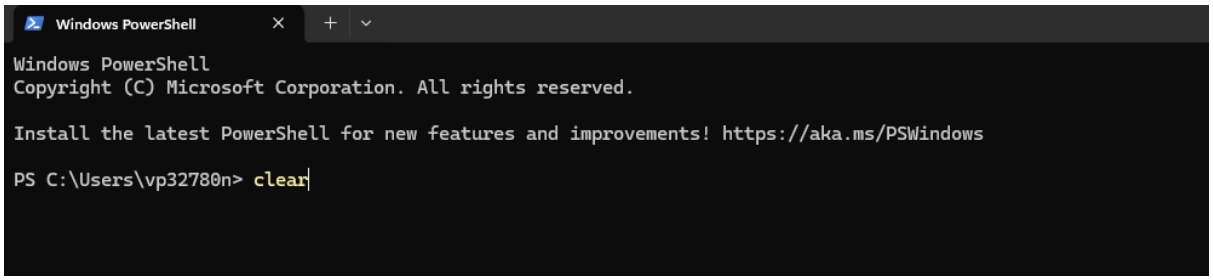

Figure 3: Using clear to reset the console display before proceeding.

Step 4 — Create a New Folder on the Desktop

I navigated to the **Desktop** directory and attempted to create a new folder for my lab files. My first attempt failed due to a typo (mkdri instead of mkdir), which returned a *CommandNotFoundException* error. After correcting the command, the folder was successfully created.

Observed result:

- The mkdri command failed, producing an error.
- The corrected mkdir command created a new directory named **Lap01** on the Desktop.
- I then changed into the new folder using cd .\Lap01\.

Why this matters:

Creating a dedicated folder ensures that all lab files are stored in one place, keeping the work organized and easy to reference later. Encountering and fixing the typo also highlights the importance of accuracy in command syntax.

Figure 4: Creating and navigating into a new folder (Lap01) on the Desktop. The first attempt failed due to a typo, corrected with mkdir

Step 5 — Verify the Logged-In User

I used the whoami command to confirm the account context under which the PowerShell session was running.

Observed result:

The output displayed pace\vp32780n, indicating that the current user is **vp32780n** under the domain **pace**.

Why this matters:

Confirming the logged-in user ensures that commands are being executed under the correct account and privileges, which is especially important for security tasks and file permissions.

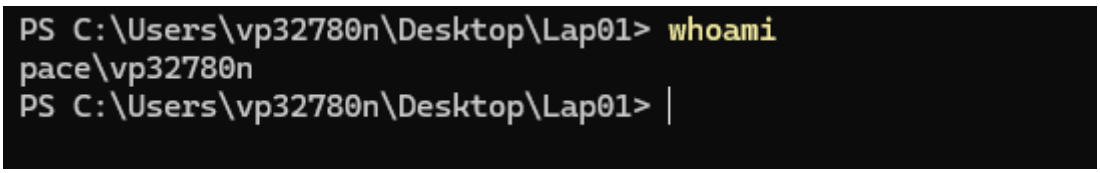

Figure 5: Using the whoami command to verify the current logged-in user account in PowerShell

Step 6 — Review Directory Contents and Confirm Current Path

I reviewed the contents of my working directory and confirmed the current path using both pwd and the Get-Location cmdlet.

Observed result:

- The dir command listed the contents of the **Lap01** directory (currently empty).
- Both pwd and Get-Location returned the same result:

  C:\Users\vp32780n\Desktop\Lap01.

Why this matters:

These commands help verify the current location in the file system. Using both pwd and Get-Location demonstrates that PowerShell provides multiple methods for confirming the working directory, which is important for accuracy in file operations.

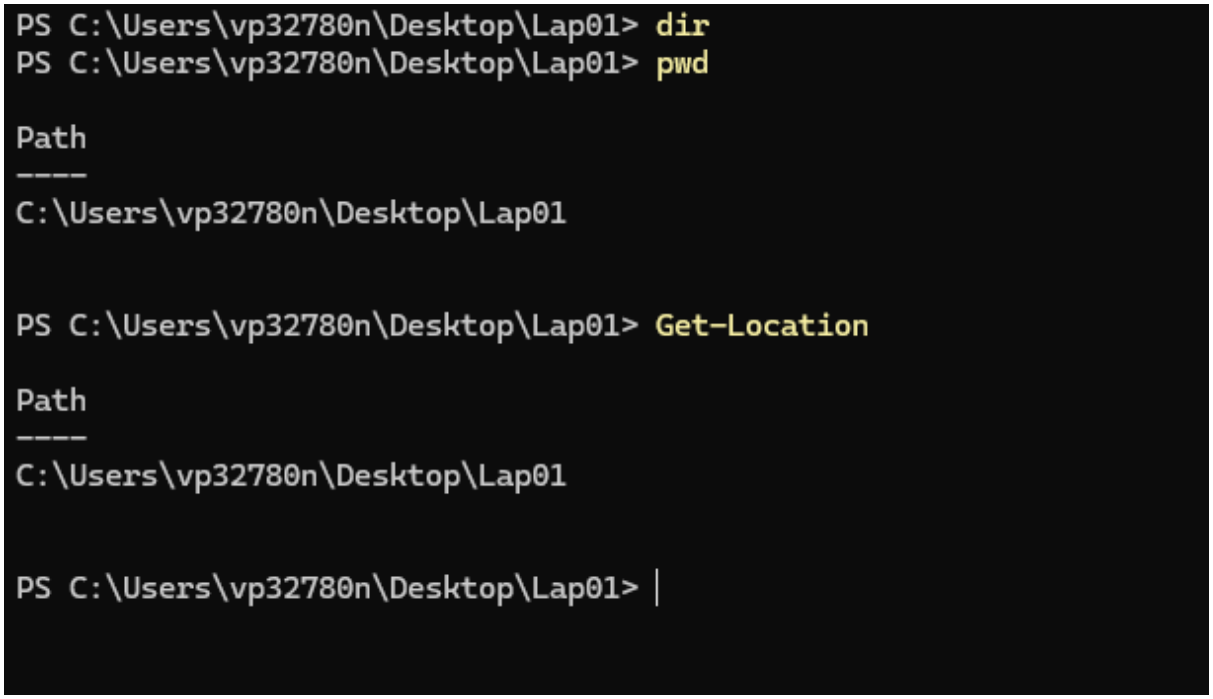

Figure 6: Using dir, pwd, and Get-Location to confirm the working directory and list contents in PowerShell.

Step 7 — Navigating with pushd and popd

I practiced using the pushd and popd commands to temporarily change directories and then return to the original location.

Observed result:

- After running pushd, the current directory C:\Users\vp32780n\Desktop\Lap01 was stored on the stack.
- I navigated one level up using cd .., which moved me to the **Desktop** directory.
- Running popd restored the previous path, returning me to C:\Users\vp32780n\Desktop\Lap01.

Why this matters:

The pushd and popd commands allow quick navigation between directories without losing track of the original path. This is useful for multitasking in different folders while ensuring you can easily return to your starting location.

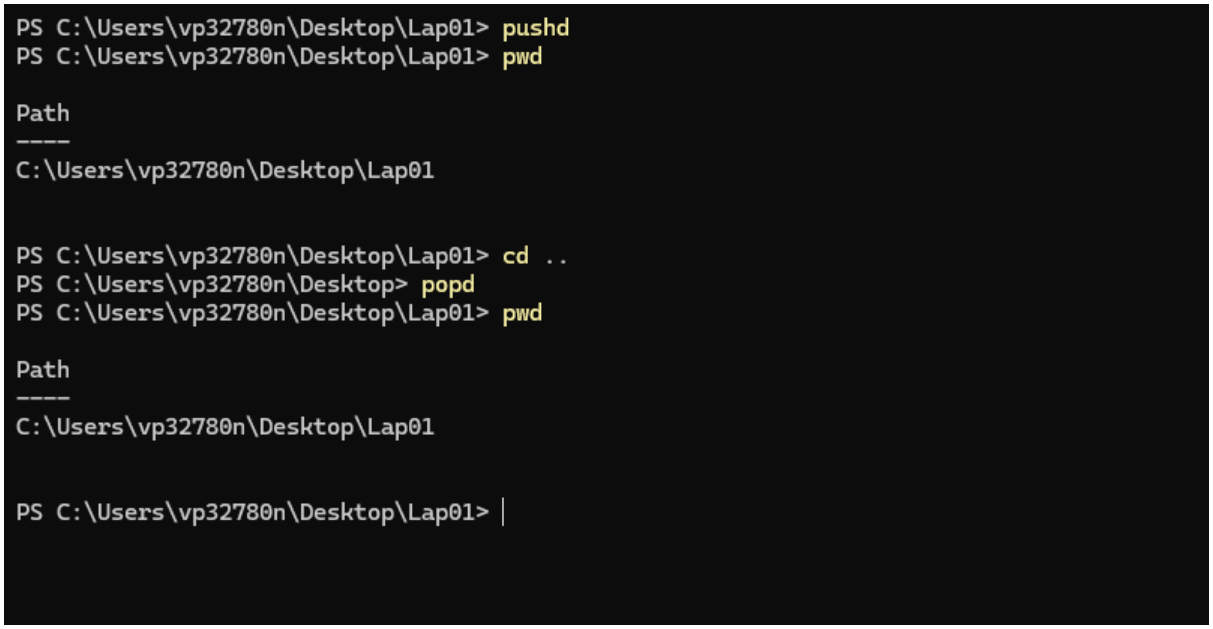

Figure 7: Demonstrating pushd and popd to temporarily navigate away from and then return to the working directory in PowerShell

Step 8 — Create a Dedicated Folder for Class Files

I created a new folder named **cyb631** inside the Lap01 directory to store all the lab-related files. I then confirmed its creation using the dir command and navigated into the folder.

Observed result:

- The mkdir cyb631 command successfully created a new directory inside Lap01.
- Running dir confirmed the presence of the **cyb631** folder.
- I then changed into the folder using cd .\cyb631\, and the prompt updated to show that I was inside the new directory.

Why this matters:

Creating a dedicated folder for lab files ensures that all work remains organized and contained within a structured workspace. This practice makes it easier to manage multiple exercises across different labs.

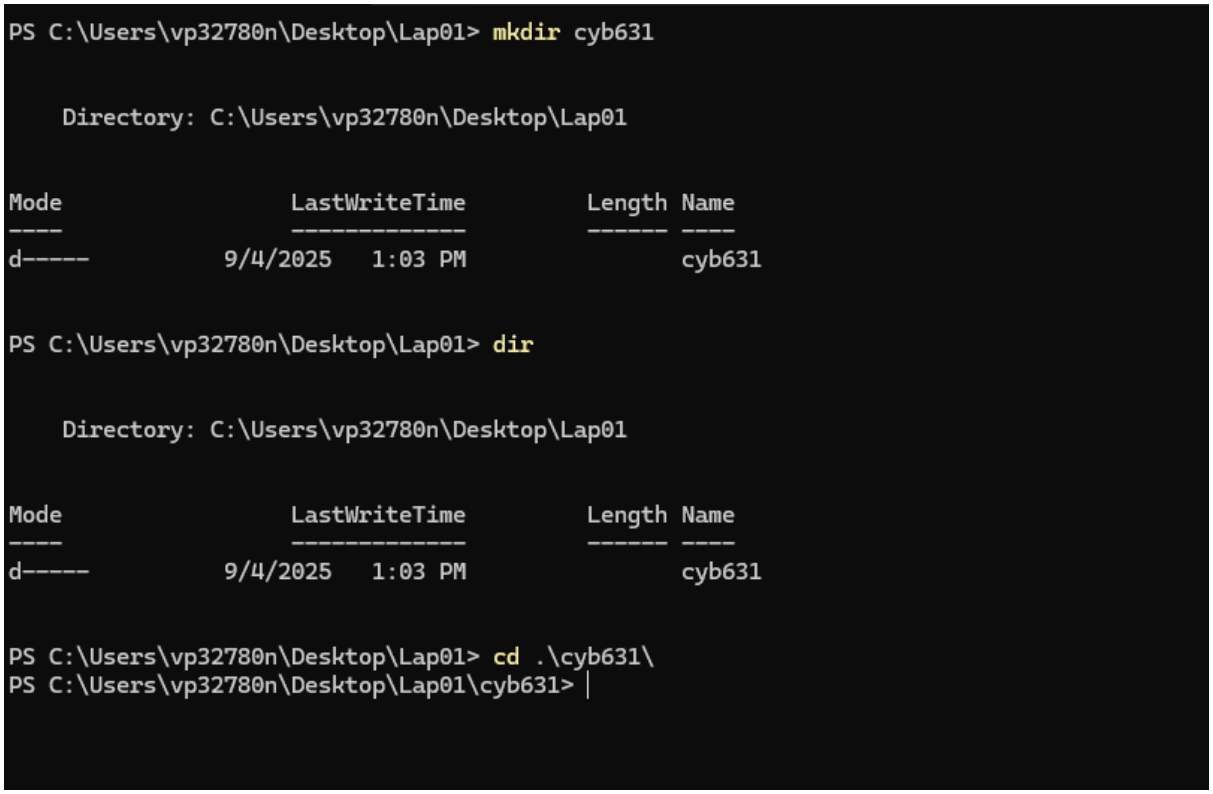

Figure 8: Creating and navigating into the new folder (cyb631) to store all Lab 1 files

Step 9 — Create and Display a New Text File

I created a new text file named **test1.txt** inside the cyb631 folder using the New-Item cmdlet. I then confirmed the file's existence with dir and displayed its contents using the cat command.

Observed result:

- The New-Item cmdlet successfully created the file **test1.txt** containing the string *Hello! World*.
- Running dir displayed the file in the directory listing.
- The cat command confirmed that the file's contents were *Hello! World*.

Why this matters:

Creating and reading files from the command line is a core skill in PowerShell. It

demonstrates the ability to automate file creation, verify existence, and inspect contents without relying on graphical tools.

Figure 9: Creating test1.txt using New-Item and confirming its contents with cat

Step 10 — Store File Path in a Variable and Access Contents

I stored the full path of the text file **test1.txt** into a PowerShell variable named $filename. I then used the Get-Content cmdlet to confirm that the variable pointed to the correct file and displayed its contents.

Observed result:

- The $filename variable was assigned the file path of test1.txt.
- Running Get-Content $filename output the text *Hello! World*, confirming the variable was correctly mapped.

Why this matters:

Storing file paths in variables simplifies command execution, especially when reusing the same file multiple times. It reduces typing errors, improves script readability, and is an essential practice for automation.

Figure 10: Storing the file path in the $filename variable and displaying file contents with Get-Content.

Step 11 — Modify File Attribute to Read-Only

I used the attrib command to change the file attribute of **test1.txt** from archive (a) to read-only (r). After applying the change, I listed the directory contents again to verify the update.

Observed result:

- Before applying the command, the file attribute was a (archive).
- After running attrib +R $filename, the attribute updated to ar, indicating the file is now both archived and read-only.
- The dir output confirmed the attribute change.

Why this matters:

Changing file attributes allows administrators to control how files can be modified. Setting a file as read-only prevents unauthorized or accidental edits, which is critical in maintaining the integrity of system or log files.

```
PS C:\Users\vp32780n\Desktop\Lap01\cyb631> dir


    Directory: C:\Users\vp32780n\Desktop\Lap01\cyb631


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          9/4/2025   1:06 PM             12 .-nametest1.txt


PS C:\Users\vp32780n\Desktop\Lap01\cyb631> attrib +R $filename
PS C:\Users\vp32780n\Desktop\Lap01\cyb631> dir


    Directory: C:\Users\vp32780n\Desktop\Lap01\cyb631


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-ar---          9/4/2025   1:06 PM             12 .-nametest1.txt


PS C:\Users\vp32780n\Desktop\Lap01\cyb631>
```

Figure 11: Changing the file attribute of test1.txt to read-only using attrib +R

Step 12 — Revert File Attribute and Verify Edit Access

I reverted the file attribute of **test1.txt** back to normal by removing the read-only flag. After that, I opened the file in Notepad to confirm that it was editable again.

Observed result:

- Running attrib -R $filename successfully removed the read-only flag, leaving only the archive (a) attribute.
- The dir command confirmed that the file no longer had the read-only (r) attribute.
- When I opened the file with notepad $filename, it displayed the text *Hello! World*, and I now had permission to modify it.

Why this matters:

This step demonstrates control over file permissions at the attribute level. Being able to toggle between read-only and writable states is important for security (protecting critical files) and for system administration tasks.

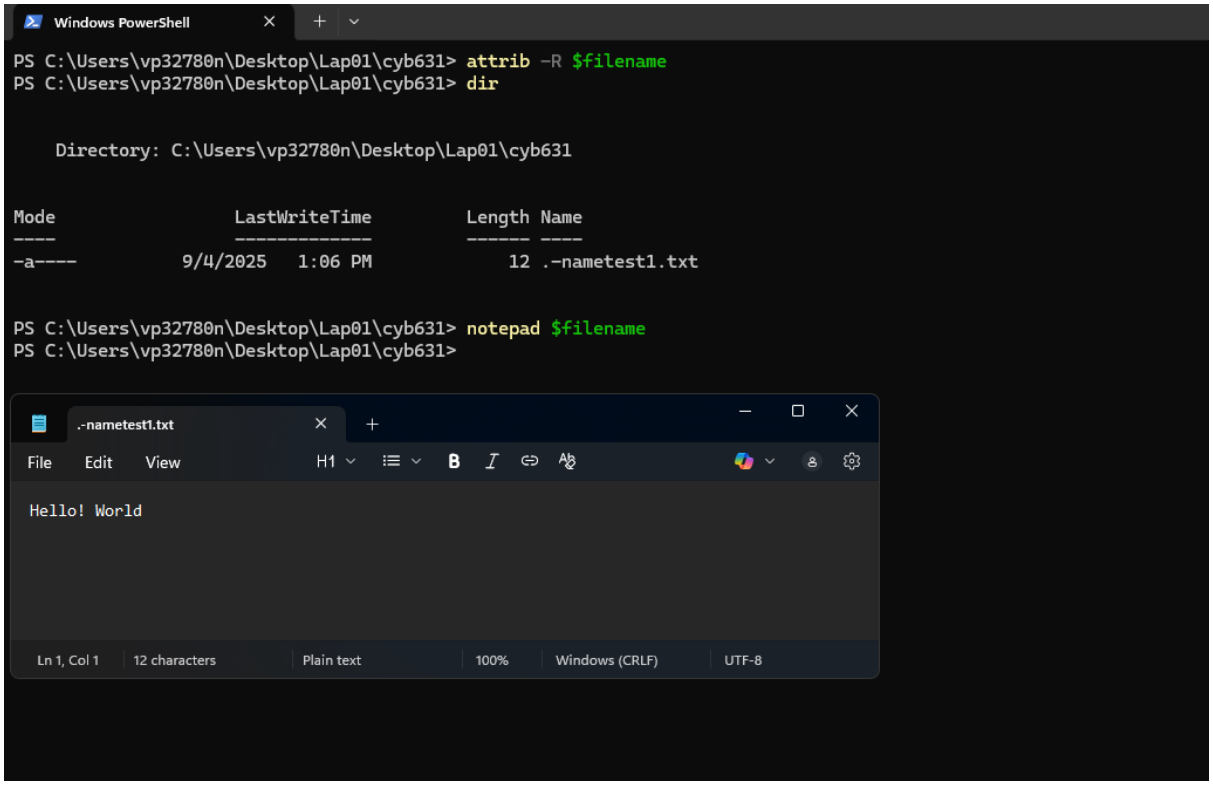

Figure 12: Removing the read-only attribute and verifying that the file can be edited again in Notepad

Exercise II: Simple Object Operation

Step 1 — Concatenate Strings into a Single Object

I created a simple string object by concatenating "Hello" and " World".

Observed result:

The output displayed Hello World, confirming that the two strings were successfully combined into one object.

Why this matters:

This step demonstrates how PowerShell processes strings as objects and allows concatenation. String manipulation is a foundational concept that will be used in more advanced scripting tasks.

Figure 13: Concatenating two strings to form a single string object in PowerShell

Step 2 — Assign String to a Variable and Inspect Member

I stored the concatenated string "Hello World" in a variable named $s1. I then used Get-Variable to confirm the assignment and Get-Member to explore the available properties and methods of the string object.

Observed result:

- The Get-Variable command confirmed that $s1 contained the value *Hello World*.
- The Get-Member output showed that $s1 is a **System.String** object. It displayed numerous methods such as ToLower(), ToUpper(), and Substring(), as well as the property Length.

Why this matters:

Exploring the properties and methods of an object is essential in PowerShell scripting. It

demonstrates that every piece of data is treated as an object, which can be queried and manipulated using built-in methods and properties.

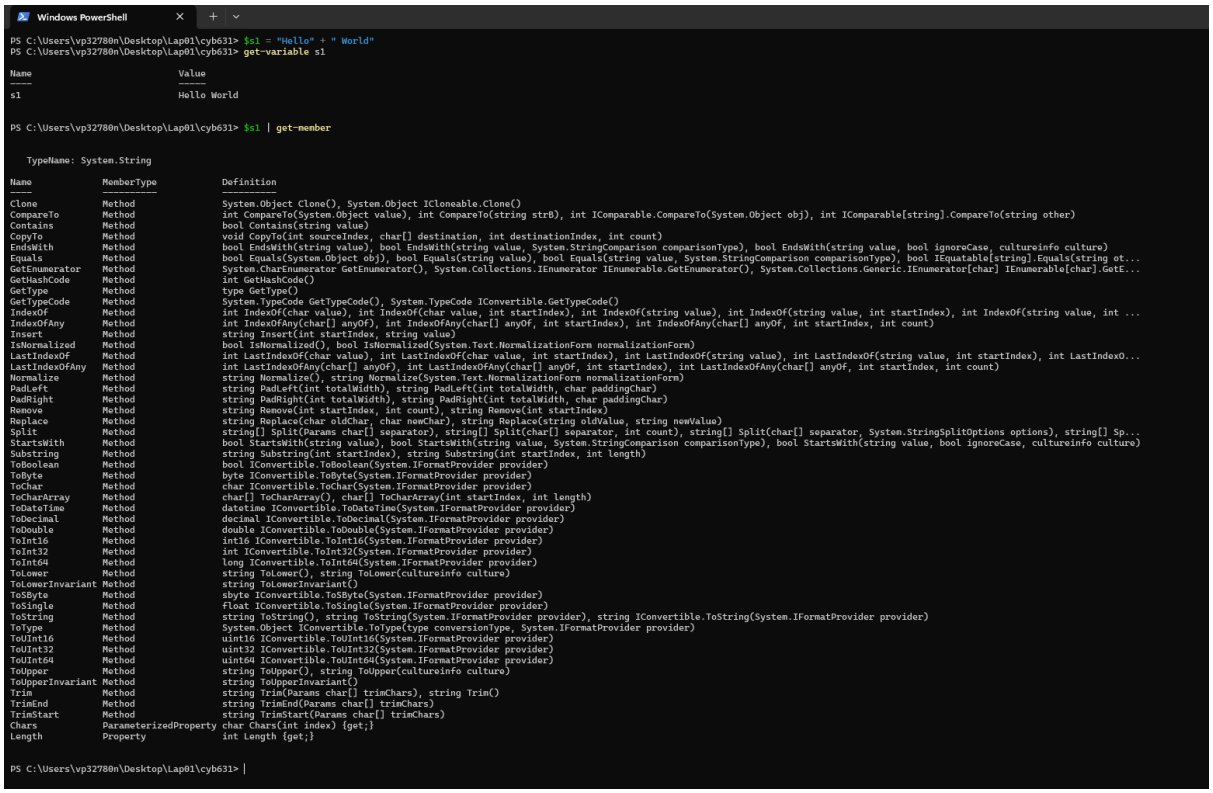

Figure 14: Assigning a string to a variable and inspecting its members with Get-Member

Step 3 — Inspect String Properties and Methods

I used the .Length property to determine the number of characters in the string stored in $s1, and then applied the .ToLower() method to convert all characters to lowercase.

Observed result:

- The .Length property returned **11**, which matches the number of characters (including the space) in "Hello World".
- The .ToLower() method converted the string to hello world, demonstrating string manipulation.

Why this matters:

This step highlights how PowerShell treats strings as objects with accessible properties and methods. Understanding how to use these features is crucial for performing data processing, formatting, and text analysis tasks in scripts.

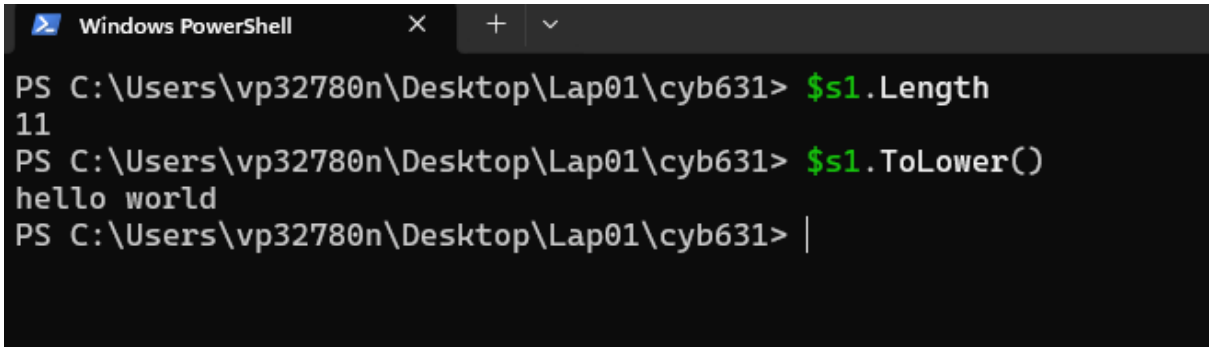

Figure 15: Using the .Length property to count characters and .ToLower() method to convert a string to lowercase

Exercise I I I: Gather Process Information

Step 1 — List All Running Processes

I ran the Get-Process cmdlet to display all currently running processes on the system.

Observed result:

The command produced a detailed table of active processes, including columns for handles, memory usage (NPM, PM, WS), CPU time, process ID, and process name. Examples of listed processes include **chrome**, **explorer**, **powershell**, and **msedge**.

Why this matters:

Viewing the list of active processes is a critical part of system monitoring and security operations. It helps administrators identify performance bottlenecks, monitor resource usage, and detect suspicious or unauthorized processes that could pose security risks.

Windows PowerShell

```
PS C:\Users\vp32780n\Desktop\Lap01\cyb631> Get-Process

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    521      25    12320      30352       5.55   11960   3 AdobeCollabSync
    437      20     8448      25932       0.20   12604   3 AdobeCollabSync
    265      15     5388      14760              5728   0 AggregatorHost
    234      17    63076      82124      20.02   21340   3 ai
    234      17    46712      58924       0.27   24688   3 ai
    234      16    22760      40768       0.39   24972   3 ai
    464      25    67104      47432       0.36    3752   3 ApplicationFrameHost
    141      11     2060       2644              16676   0 armsvc
    497      36    16972      53884       0.22   18940   3 backgroundTaskHost
    610      44    85480      82588       1.83   16848   3 CalculatorApp
    386      32    63548     109816       4.73    1260   3 chrome
    367      28    58796     102548       2.06    1436   3 chrome
    394      28    86168     129092       2.64    2020   3 chrome
    221      19    13664      31516       0.06    2476   3 chrome
    480      39   140476     215344      24.61    6288   3 chrome
    899      60   420120     286748     157.77    9316   3 chrome
    253      10     2500       9580       0.11   11672   3 chrome
    216      16    12328      22972       1.36   11776   3 chrome
    441      35    56736      87592      57.19   14044   3 chrome
    340      30    65120      98472      16.75   15224   3 chrome
    430      56   517500      74516      10.00   15848   3 chrome
    259      17     8416      23412       0.77   16232   3 chrome
   2196      81   249044     348916     320.38   16524   3 chrome
    533      44   251200     327416     312.47   20132   3 chrome
    539      31    81836     134968       4.19   20576   3 chrome
    341      27    66404     116532       0.97   20676   3 chrome
    481      24    70236      42332       0.22   21624   3 chrome
    347      27    28896      59192       0.28   22080   3 chrome
    370      29   101696     130768      10.67   23988   3 chrome
    392      29    39700      80828       2.09   24672   3 chrome
    458      30   112164     150500       4.84   25532   3 chrome
    137       9     1676      10688       0.02   20868   3 conhost
    489      26    67248      53700       0.50    5588   3 CrossDeviceResume
    764      24     2408       4012              1008   0 csrss
    756      26     3480       8084              6932   3 csrss
    555      22    10748      38684      16.17    6992   3 ctfmon
    269      19     5000       4180              8372   0 CTskMstr
    354      22    11328      41048       0.31   23420   3 DataExchangeHost
   4187      53    31152      24380              3936   0 DefendpointService
    286      16     4748       4732             11384   0 dllhost
   1804      77   184364      97908              9992   3 dwm
    507      32    44140      18836              3968   0 Examsoft.SoftShield
   4992     133   240196     309604      56.92   18212   3 explorer
     43       7     1952        992              1372   0 fontdrvhost
     43       8     4192       9488             18004   3 fontdrvhost
      0       0       60          8                 0   0 Idle
    187      12     2524       5936              2868   0 igfxCUIService
    335      16     4016      19632       0.31    7772   3 igfxEM
    163      10     1964       2840               952   0 IntelCpHDCPSvc
    149       9     2020       2832              2232   0 IntelCpHeciSvc
    145      10     1660       2700              3952   0 jhi_service
   1791      35    11384      29236              1204   0 lsass
      0       0      976     304356              2788   0 Memory Compression
    218      15     2384       5700              7892   0 MicrosoftEdgeUpdate
    523      18    12156      20796              3156   0 MpDefenderCoreService
    248      15     5480       3296             15184   0 msdtc
    242      14     8948      27788       0.22   2988   3 msedge
    171      11     8352      19248       0.19   5364   3 msedge
    354      21    15216      45304       1.56   5688   3 msedge
    202      17    15400      31432       0.08   6580   3 msedge
    185      10     2476      10096       0.08   7368   3 msedge
    310      24    57564     113484      10.00   8004   3 msedge
    283      26    60908      59868       0.91   9920   3 msedge
    298      22    21232      63692       0.41   14180  3 msedge
    598      36   150124      94432      10.02   16008  3 msedge
   1691      64    81136     189764      25.70   17348  3 msedge
```

Figure 16: Displaying all active processes using the Get-Process cmdlet in PowerShell

Step 2 — Filter Processes Starting with "W"

I used the gps alias with the -n parameter to display only the processes whose names start with the letter **W**.

Observed result:

The command output displayed only processes beginning with "W," including **WavesSvc64**, **WindowsTerminal**, **WINWORD**, **WmiPrvSE**, and **Winlogon**. The table provided details such as handles, memory usage, CPU utilization, process IDs, and session information for each filtered process.

Why this matters:

Filtering processes allows administrators to quickly locate specific applications or services of interest without needing to search manually through all active processes. This technique is especially useful for targeted troubleshooting and performance monitoring.

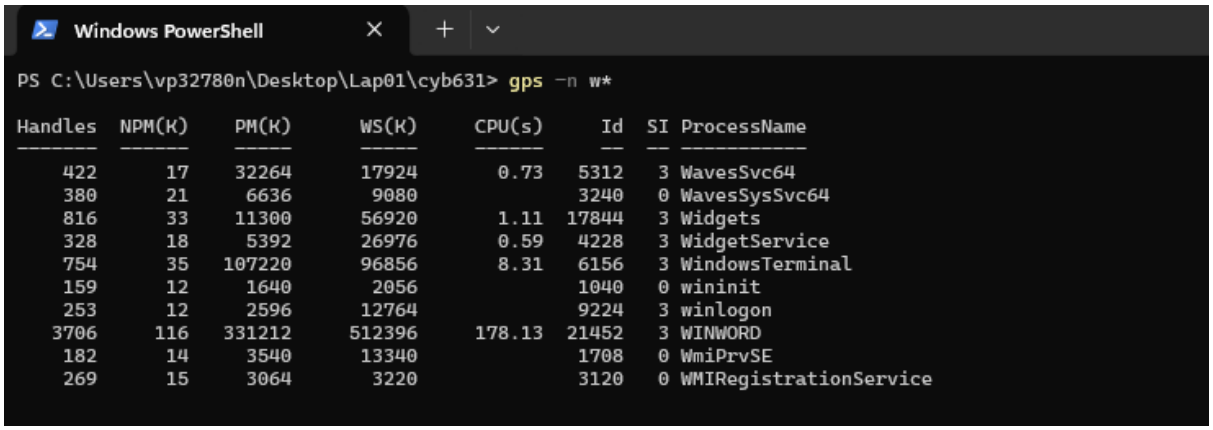

*Figure 17: Filtering active processes that start with the letter "W" using gps -n w**

Step 3 — Store Notepad Process in a Variable and Attempt to Kill It

I launched Notepad and stored its process information in a variable named $prs. After verifying that the variable contained the Notepad process, I attempted to terminate it using the .Kill() method.

Observed result:

- The $prs variable successfully stored the active Notepad processes.

- When I executed $prs.Kill(), the selected Notepad instance was terminated.
- Running PowerShell as Administrator allowed the kill command to succeed without an "Access Denied" error.

Why this matters:

This step shows how PowerShell can be used not only to monitor but also to control processes directly. The ability to end processes is essential for system administration and security incident response, particularly when dealing with frozen or suspicious applications.

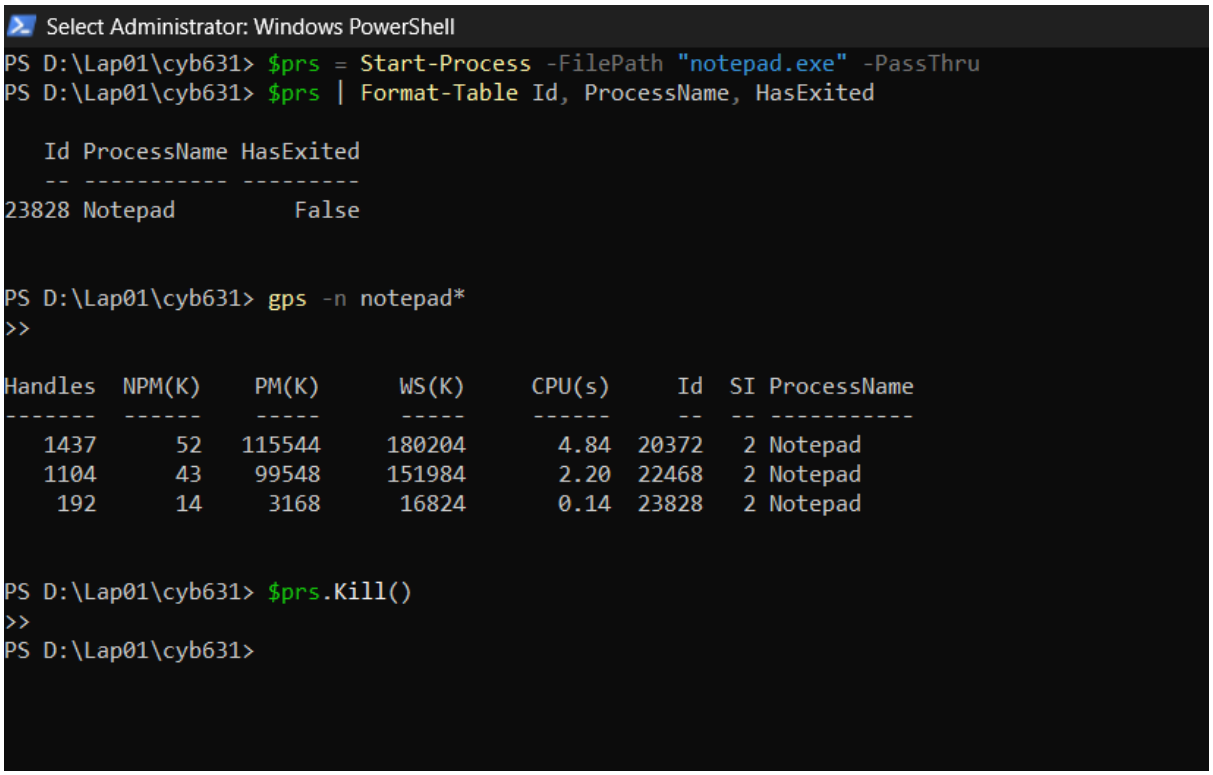

Figure 18: Storing the Notepad process in a variable and terminating it with $prs.Kill()

Step 4 — Show Processes with ID ≥ 10,000 (Pipeline, Sorted Table)

I used a pipeline to filter running processes whose **Id** is greater than or equal to 10,000, sorted them by Id, and formatted the output as a concise table.

Observed result:

A table listing only high-ID processes (e.g., svchost, chrome, LockApp, ShellExperienceHost, etc.) with their **Id**, **Name**, **Handles**, and **Description** columns displayed in sorted order.

Why this matters:

Chaining cmdlets with the pipeline lets you filter, sort, and present process data efficiently—exactly the pattern used in real incident response and performance triage.

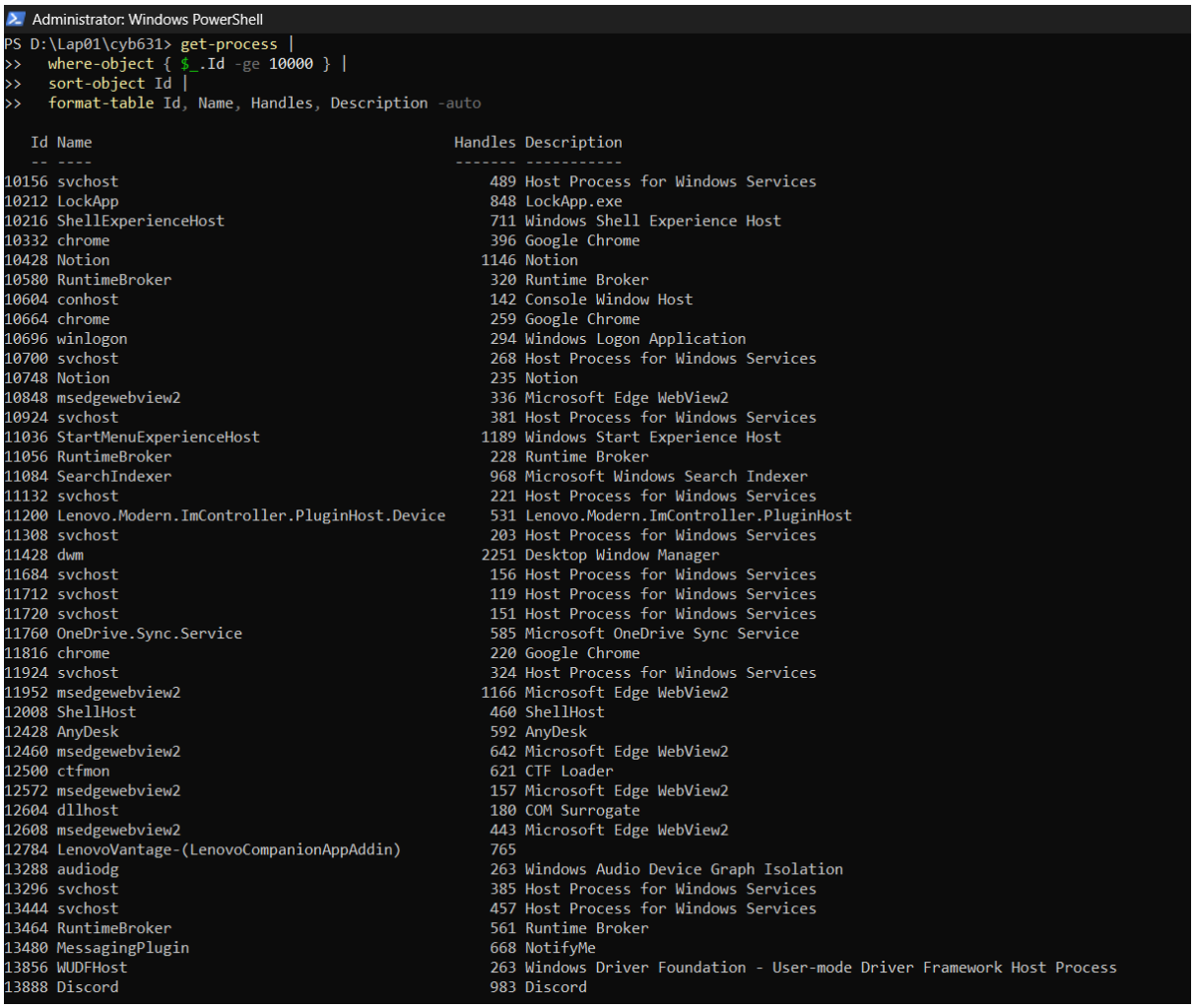

Figure 19: Filtering to processes with Id ≥ 10,000 and formatting the results as a table

Exercise IV: Useful PowerShell Commands

Step 1 — Explore Help and List Available Commands

I first ran the Get-Help Get-Process cmdlet to view available documentation for the **Get-Process** command. PowerShell prompted to update the help files, which provides the most current documentation from Microsoft.

Next, I used the Get-Command cmdlet to list all available commands in the current session.

Observed result:

- Get-Help Get-Process displayed help content describing the purpose of the cmdlet, syntax, and available options.
- Get-Command produced a list of available commands, including their command type (e.g., Alias, Cmdlet), name, version, and source module.

Why this matters:

The Get-Help cmdlet provides detailed documentation for understanding how specific commands work, while Get-Command is essential for discovering all the tools available in a session. Together, they are critical for learning and troubleshooting in PowerShell.

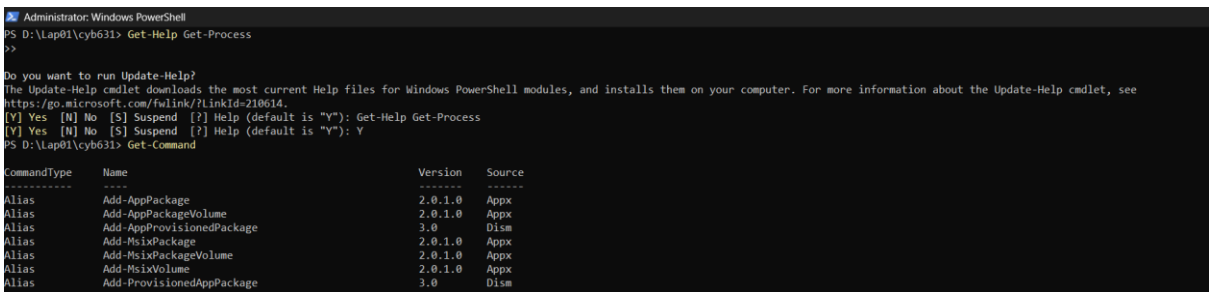

Figure 20: Using Get-Help Get-Process to display documentation and Get-Command to list all available commands

Step 2 — Explore Members of the Get-Process Object

I piped the output of the Get-Process cmdlet into Get-Member to explore all the properties, methods, and events associated with the **System.Diagnostics.Process** object.

Observed result:

The output identified the object type as **System.Diagnostics.Process** and listed its available members, such as:

- **AliasProperties**: Handles, Name, NPM (NonpagedMemorySize64), PM (PagedMemorySize64), SI (SessionId), VM (VirtualMemorySize64), WS (WorkingSet64).
- **Events**: Disposed, ErrorDataReceived, Exited, OutputDataReceived.
- **Methods**: Close(), Kill(), Start(), WaitForExit(), BeginOutputReadLine(), among many others.

Why this matters:

This step demonstrates PowerShell's object-oriented design. Understanding the properties and methods of process objects enables administrators to monitor, control, and manipulate running processes directly from the command line.

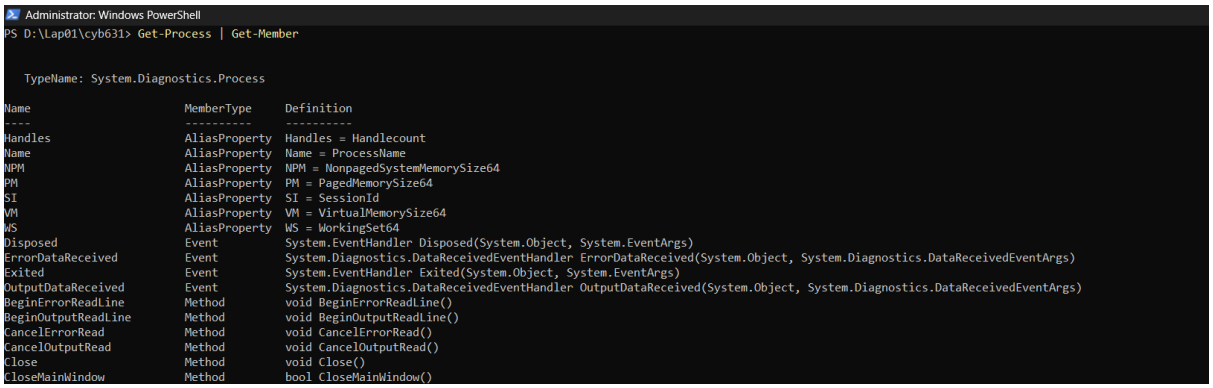

Figure 21: Exploring properties, methods, and events of the System.Diagnostics.Process object using Get-Member

Exercise V: First PowerShell Script

Step 1 — List Processes Beginning with "N"

I used the Get-Process cmdlet with the -n parameter to display all processes whose names begin with the letter **N**. This prepares the dataset that will later be used to calculate the sum of their handles.

Observed result:

The output displayed all processes starting with the letter "N," such as **Notepad**, **NisSrv**, or other system services depending on what was running at the time. Each process included information such as handles, memory usage, CPU time, and process ID.

Why this matters:

This step demonstrates filtering processes by name, which is an important capability in PowerShell. The filtered list becomes the foundation for summing handle counts in later scripting steps, showing how command output can be built into more advanced scripts.

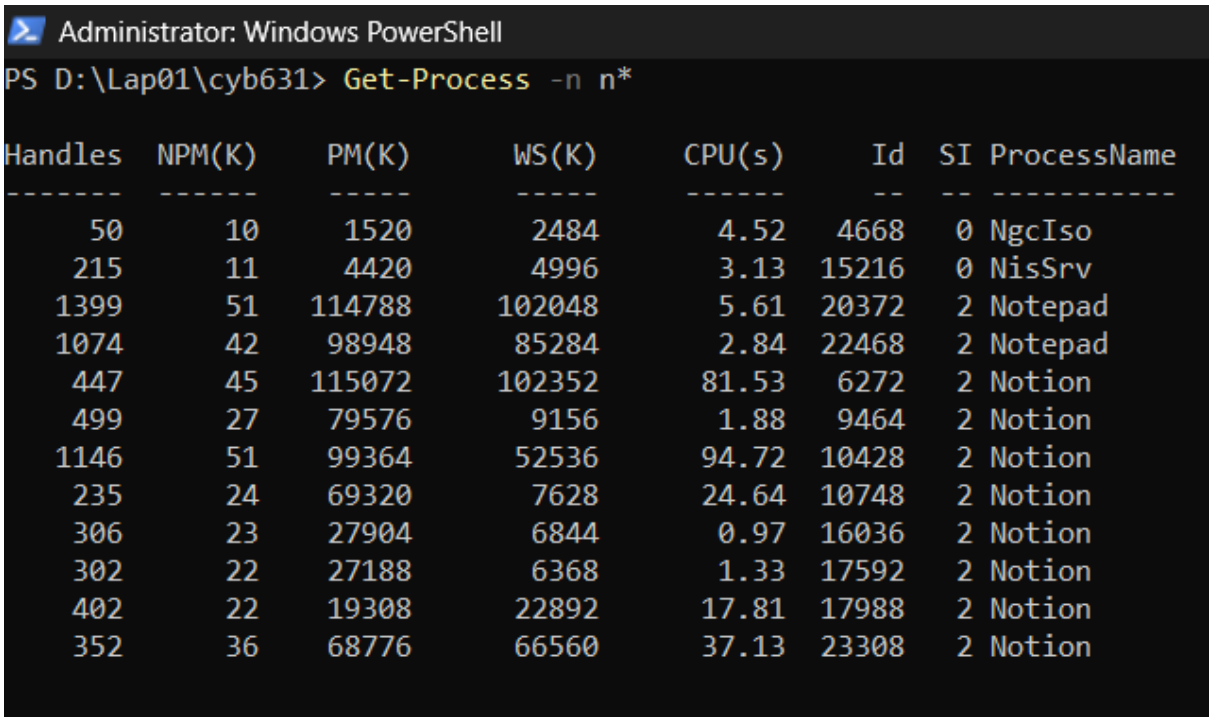

*Figure 22: Listing processes that begin with the letter "N" using Get-Process -n n**

Step 2 — Sum Handles of "N" Processes

I initialized a counter variable and then used a foreach loop to iterate through all processes whose names start with the letter "n." For each process, I added its handle count to the $hcount variable. Finally, I displayed the total and verified it using the Measure-Object cmdlet.

Observed result:

- The total number of handles for processes beginning with "n" was **6425**.
- The result from the foreach loop matched the value obtained using Measure-Object, confirming the correctness of the script.

Why this matters:

This step demonstrates the power of scripting in PowerShell. By combining loops and variables, I automated the aggregation of process data. Using Measure-Object as a verification method highlights how PowerShell provides multiple approaches to accomplish the same task.

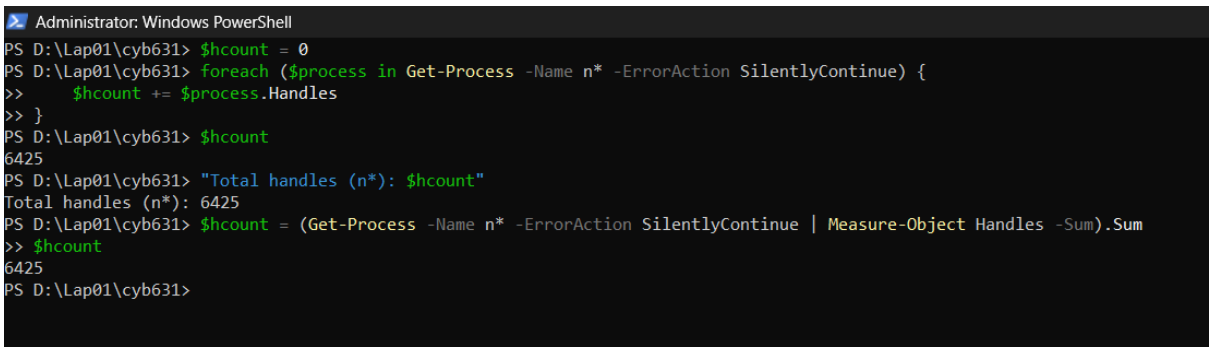

Figure 23: Summing the handles of all processes beginning with "n" using a foreach loop and verifying with Measure-Object

Step 3 — Run the script and manage the execution policy

I ran my script (script1.ps1) and adjusted the execution policy *for this session* to permit script execution, then reviewed the effective policies at each scope.

Observed result:

- With -Scope Process Bypass, the script executed successfully and printed the total handle count (e.g., **6415**).
- Attempts to change -Scope CurrentUser returned **PermissionDenied** because a higher-scope policy overrides it.
- Get-ExecutionPolicy -List showed Process = Bypass (effective for this session), while other scopes (MachinePolicy, UserPolicy, LocalMachine) remained unchanged.

Why this matters:

Understanding execution policies is essential for safe automation. Using **Process-scoped Bypass** lets you run trusted scripts temporarily **without** weakening system-wide policy, which aligns with least-privilege and secure-by-default practices.

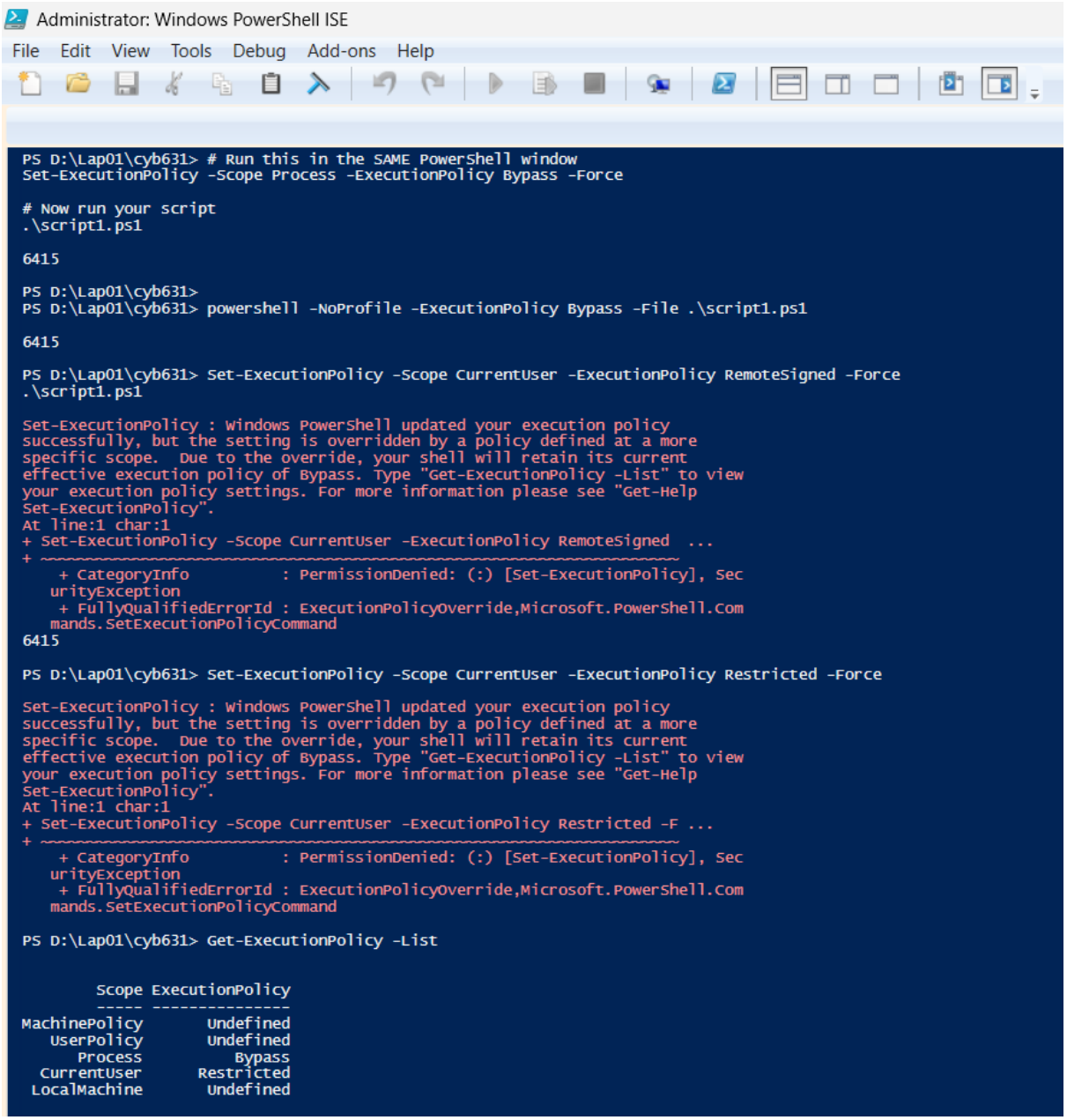

Figure 24: Running script1.ps1 with a process-scoped policy bypass, failed attempts to change user-scope policy due to higher-scope overrides, and effective policies via Get-ExecutionPolicy -List

Step 4 — Create and Run script1.ps1

I opened **PowerShell ISE** (Integrated Scripting Environment) and created a new script file named script1.ps1. Inside the file, I entered the three commands to sum the handles of processes beginning with the letter “n”

I then saved the script and attempted to run it. At first, execution was blocked by the system's execution policy.

Observed result:

- The script executed successfully and returned the total handle count (e.g., **6415**).
- Adjusting the execution policy at the **Process** scope allowed me to run the script temporarily without changing system-wide settings.

Why this matters:

Creating and running a .ps1 file demonstrates how PowerShell commands can be combined into reusable scripts. This is the foundation for automation, enabling repeatable tasks while respecting security controls like execution policies.

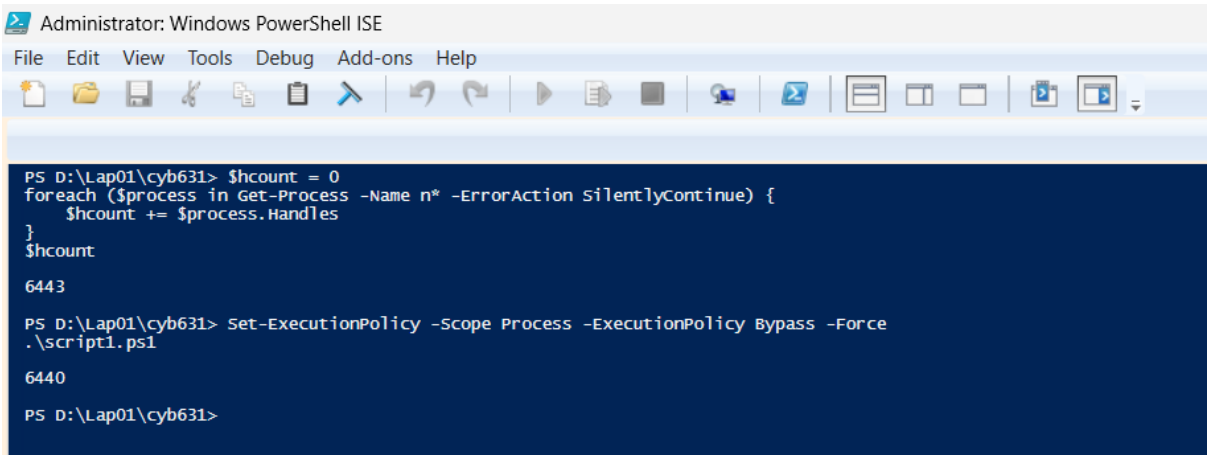

Figure 25: Saving the handle-summing commands into script1.ps1 and executing it successfully after adjusting the execution policy

Exercise VI: Develop Your Own PowerShell Script – CPU Time

Step 1 — Create a Script to Identify Top CPU-Consuming Processes

I wrote a PowerShell script named **cputime.ps1** to sort processes by CPU usage and display the top five consumers along with their process IDs and names.

Observed result:

When executed, the script displayed the **five processes using the most CPU time**, showing each process's **ID, name, and CPU usage**. Example processes included chrome, explorer, and MsEdge, depending on what was active during execution.

Why this matters:

Identifying CPU-heavy processes is critical for performance monitoring and threat detection. A legitimate system process consuming high CPU may indicate inefficiency, while unexpected processes with high usage may signal malware or system misconfiguration.

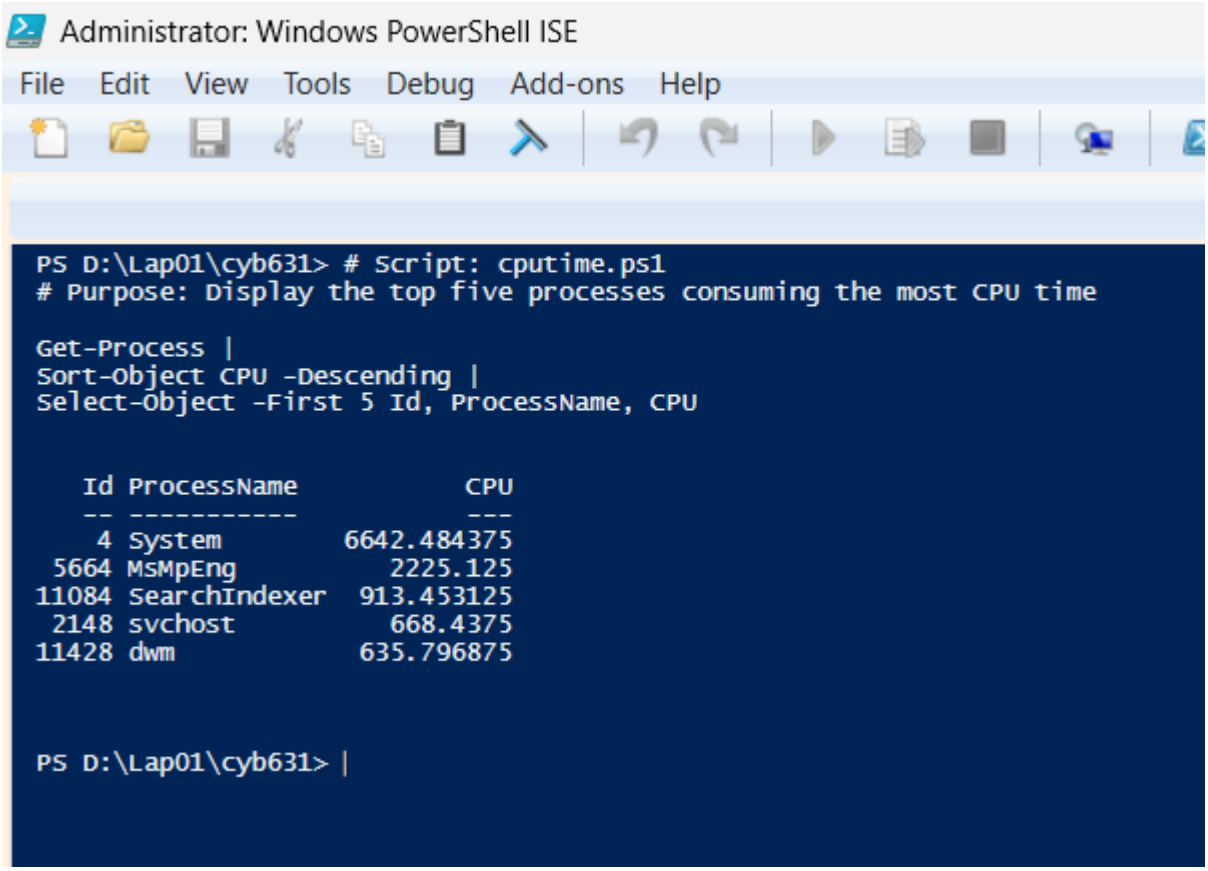

Figure 26: Running cputime.ps1 to display the top five CPU-consuming processes.

Exercise VII: Lab and Class Reflection

What I liked about this lab:

I appreciated how the lab combined both foundational and practical aspects of PowerShell. It guided me from simple commands like whoami, dir, and pwd to creating scripts that manipulate processes and monitor CPU usage. The hands-on approach made it easier to connect theory with real-world system administration and security tasks.

Challenges I encountered:

The main challenges were managing execution policies and understanding error messages. For example, when changing execution policies at the user scope, I encountered permission errors due to higher-scope overrides. Initially, this was confusing, but through experimentation I learned how to use process-scoped bypass policies securely.

Suggestions for improving the class:

It would be helpful to include more examples that directly relate PowerShell usage to cybersecurity scenarios, such as log analysis, malware detection, or automated incident response. Additional practice exercises on string parsing and file manipulation would also reinforce scripting skills.

Turning off virtual machines:

As instructed, I shut down the Windows virtual machine after completing the lab to free up server resources.

References

Holmes, L. (2021). *Windows PowerShell cookbook* (4th ed.). O'Reilly Media.

Microsoft. (2023, June 12). *Table of basic PowerShell commands*. Microsoft DevBlogs. https://devblogs.microsoft.com/scripting/table-of-basic-powershell-commands/

Pace University. (2023). *CYB 631: Automating Information Security with Python and Shell Scripting – Lab 1: PowerShell basics and gathering host information* [Course handout]. Department of Computer Science, Pace University.

The White Hat Oath & Code of Ethics
Pace University

(Special thanks to Michael Whitman and Avi Rubin for providing the source of this agreement)

This is a working document that provides further guidelines for Information Assurance courses taught at Pace University. If you have questions about any of these guidelines, please contact the course instructors. When in doubt, the default action should be to ask the instructors.

1) The course project may require technical means of discovering information about others with whom you share a computer system. As such, non-technical means of discovering information are disallowed (e.g., posing as system administrator over the phone to ask for user passwords).

2) ANY data that is stored outside of the course accounts can be used only if it has been explicitly and intentionally published, (e.g. on a web page), or if it is in a publicly available directory, (e.g. /etc, /usr).

3) Gleaning information about individuals from anyone outside of the course is disallowed.

4) Impersonation, e.g. forgery of electronic mail, is disallowed.

5) If you discover a way to gain access to any account other than your own (including root), do NOT access that account, but immediately inform the course instructors of the vulnerability. If you have inadvertently already gained access to the account, IMMEDIATELY exit the account and inform the course instructors.

6) All explorations should be targeted specifically to the assigned course accounts. ANY tool that indiscriminately explores non-course accounts for vulnerabilities is specifically disallowed.

7) Using the web to find exploration tools and methods is allowed. In your reports, provide full attribution to the source of the tool or method.

8) If in doubt at all about whether a given activity falls within the letter or spirit of the course exercise, discuss the activity with the instructors BEFORE exploring the approach further.

9) You can participate in the course exercise only if you are registered for a grade in the class. ANY violation of the course guidelines may result in disciplinary or legal action.

10) Any academic misconduct or action during the course of the class can result in failure of and dismissal from the course.

Code of Ethics

Code of Ethics Preamble: (Source www.isc2.org Code of ethics)
Safety of the commonwealth, duty to our principals, and to each other requires that we adhere, and be seen to adhere, to the highest ethical standards of behavior.
Therefore, strict adherence to this code is a condition of laboratory admission.

Code of Ethics Canons:
Protect society, the commonwealth, and the infrastructure.
Act honorably, honestly, justly, responsibly, and legally.
Provide diligent and competent service to principals.
Advance and protect the profession.

The following additional guidance is given in furtherance of these goals.

Objectives for Guidance

Protect society, the commonwealth, and the infrastructure
Promote and preserve public trust and confidence in information and systems.
Promote the understanding and acceptance of prudent information security measures.
Preserve and strengthen the integrity of the public infrastructure.
Discourage unsafe practice.

Act honorably, honestly, justly, responsibly, and legally
Tell the truth; make all stakeholders aware of your actions on a timely basis.
Observe all contracts and agreements, express or implied.
Treat all constituents fairly. In resolving conflicts, consider public safety and duties to principals, individuals, and the profession in that order.
Give prudent advice; avoid raising unnecessary alarm or giving unwarranted comfort. Take care to be truthful, objective, cautious, and within your competence.
When resolving differing laws in different jurisdictions, give preference to the laws of the jurisdiction in which you render your service.

Provide diligent and competent service to principals
Preserve the value of their systems, applications, and information.
Respect their trust and the privileges that they grant you.
Avoid conflicts of interest or the appearance thereof.
Render only those services for which you are fully competent and qualified.

Advance and protect the profession
Sponsor for professional advancement those best qualified. All other things equal, prefer those who are certified and who adhere to these canons. Avoid professional association with those whose practices or reputation might diminish the profession.
Take care not to injure the reputation of other professionals through malice or indifference.
Maintain your competence; keep your skills and knowledge current. Give generously of your time and knowledge in training others.

White Hat Agreement

As part of this course, you may be exposed to systems, tools and techniques related to Information Security. With proper use, these components allow a security or network administrator better understand the vulnerabilities and security precautions in effect. Misused, intentionally or accidentally, these components can result in breaches of security, damage to data or other undesirable results.

Since these lab experiments will be carried out in part in a public network that is used by people for real work, you must agree to the following before you can participate. If you are unwilling to sign this form, then you cannot participate in the lab exercises.

Student Agreement Form

I agree to:

- only examine the special course accounts for privacy vulnerabilities (if applicable).
- report any security vulnerabilities discovered to the course instructors immediately, and not disclose them to anyone else.
- maintain the confidentiality of any private information I learn through the course exercise.
- actively use my course account with the understanding that its contents and actions may be discovered by others.
- hold harmless the course instructors and Pace University for any consequences of this course.
- abide by the computing policies of Pace University and by all laws governing use of computer resources on campus.

I agree to NOT:

- attempt to gain root access or any other increase in privilege on any Pace University computers.
- disclose any private information that I discover as a direct or indirect result of this course exercise.
- take actions that will modify or deny access to any data or service not owned by me.
- attempt to perform any actions or use utilities presented in the laboratory outside the confines and structure of the labs.
- utilize any security vulnerabilities beyond the target accounts in the course or beyond the duration of the course exercise.
- pursue any legal action against the course instructors or Pace University for consequences related to this course.

Moreover, I consent for my course accounts and systems to be examined for security and privacy vulnerabilities by other students in the course, with the understanding that this may result in information about me being disclosed (if applicable).

This agreement has been explained to me to my satisfaction. I agree to abide by the conditions of the Code of Ethics and of the White Hat Agreement.
Signed, **Vijaysingh Puwar** Date: 09/4/2025
Printed name: **Vijaysingh Puwar** E-mail address: **vp32780n@pace.edu**

CYB631: Automating Information Security with Python and Shell Scripting

Summer 2023

Pace University

Student Information Sheet

Name: Vijaysingh Puwar

Email: vp32780n@pace.edu

Job Title & Employer (optional, if employed full-time): Graduate Student, Pace University (previously System Engineer – R. S. Infotech)

Major: MS/CYB

Year (1, 2, 3, 4), Graduation Date: Year 1, December 2026

1. Computer Knowledge and Skill Level

(a) Operating Systems:

- Windows — advanced
- Linux — intermediate
- Mac — novice
- Other — N/A

(b) Networking:

- Active Directory — intermediate
- Linux Networking — intermediate
- TCP/IP — advanced
- Other — N/A

(c) Programming:

- PowerShell — novice
- Linux shell scripting — intermediate
- Python — intermediate
- Java — intermediate
- C — novice
- Other — N/A

2. Work Experience in Networking and/or Security

- System Engineer, R. S. Infotech (2023–2024): Performed system administration, troubleshooting, and basic security monitoring.
- CCNA Certified: Designed and configured Packet Tracer labs covering VLANs, EtherChannel, STP, and inter-VLAN routing.
- Security Projects: Completed an OSINT case study on Cloudflare; developed Splunk SIEM detections; conducted penetration testing using Metasploit, Burp Suite, and Nmap.

3. Grade Expectation and Learning Goals

I expect to achieve an A grade in this course. My goal is to strengthen my PowerShell and shell scripting skills for cybersecurity automation, particularly in collecting and analyzing host information. I also aim to apply these skills to real-world scenarios such as incident response, detection engineering, and system administration.