Explore PowerShell

To complete the labs for this class you will need to create a virtual machine as described in the lab environment setup instructions [here](https://onedrive.live.com/view.aspx?resid=5EFC811CDEC9D7F0!9867&ithint=file%2cdocx&authkey=!AFmEtJQN-wdNXmA).

In this lab, we will explore the PowerShell command shell, the place where you can run scripts and interactively enter commands. PowerShell is easy to use when you understand the basic building blocks and how they work together.

Start the PowerShell application using the shortcut on the desktop of your lab VM.

Graphical user interface, text, application, website

Description automatically generated

Starting PowerShell this way starts PowerShell with normal user privileges (not admin privileges). Start a second PowerShell command shell by right-clicking on the PowerShell desktop shortcut and choosing “Run as Administrator”. Click “yes” when prompted to allow PowerShell to start with admin privileges.

Graphical user interface, text

Description automatically generated

Notice that the PowerShell application that you “Ran as Administrator” has the word “Administrator” in the title. There are several times throughout the labs that you will need to use an administrative PowerShell prompt, so pay close attention to the instructions and the screenshots.

# Variables

Let’s explore some of the features of the PowerShell command shell. Variables in PowerShell start with a **$** sign. Try defining some variables and then echoing them back onto the screen either by entering only the variable name and pressing enter or by using the **Write-Host** function.

Graphical user interface, text, application

Description automatically generated

PowerShell has many predefined variables built-in, and it can seem like magic when you see them used. For example, we can refer to the **$PSVersionTable** variable to view information about the current PowerShell version we are using.

Graphical user interface, text

Description automatically generated

We are using PowerShell version 5.1 in class. This is the version that comes installed by default on Windows 10 and 11.

To view all of the variables we have available to use, including built-in variables and any that we may have defined, you can use the **Get-Variable** function.

A picture containing text

Description automatically generated

The list below shows some of the more interesting variables you are likely to make use of.

* **Home** - The current user’s home directory
* **Null** - compare your own variables to $null to test if they are undefined.
* **Profile** - The path to a custom PowerShell script that will be loaded every time you start PowerShell
* **PWD** - “Print Working Directory” will display the path of the folder that you are currently working from.
* **PID** - The process ID of your current PowerShell session.

Text

Description automatically generated

Note: PowerShell is a case-insensitive language, and you can refer to functions and variables using any combination of upper and lowercase letters. For example, enter $PWD, $pWd or $pwd will all have the same result.

The **cd** function is an alias for the **Set-Location** function in PowerShell. Notice how it was used to return the **$home** directory without having to type **C:\Users\IEUser**, which is a nice shortcut.

# Session History

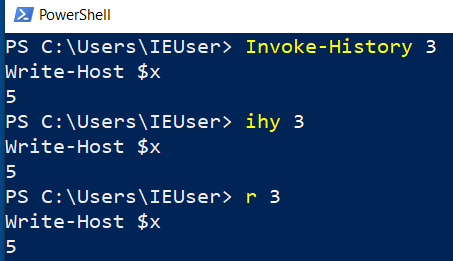
For convenience, the PowerShell command prompt also maintains a list of all the commands you have run in the current session. You can access this list using the **Get-History** function (or just **h** for short).

Graphical user interface, text

Description automatically generated

The session history is only maintained for the current PowerShell session (think of the current command shell window you are using). If you start a new PowerShell window, the session history will be blank.

Use the **Invoke-History** command to execute a specific line number from your session history. The **ihy** and **r** commands are shortcuts, or aliases, for the Invoke-History Command.



Start a PowerShell window and run the **Get-History** command. Do you see any session history? No, the session in this case will be empty because we have started a new session. If we are wanting to reference commands from our command history across sessions, or even computer restarts, we need to use the file-base history as discussed next.

# History File

In addition to the session history, there is also a file-based history that persists from session to session. This file-based history is provided by a PowerShell module called **PSReadline**. The PSReadline module is installed by default with PowerShell. This module supports to use of the up and down arrow keys to iterate through the history.

Text

Description automatically generated with medium confidence

We can view and set the configuration for the PSReadline module using the **Get-PSReadlineOption** and **Set-PSReadlineOption** commands.

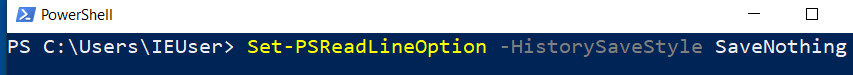
Text

Description automatically generated with medium confidence

You can use the following command to print the contents of the history file to the screen.

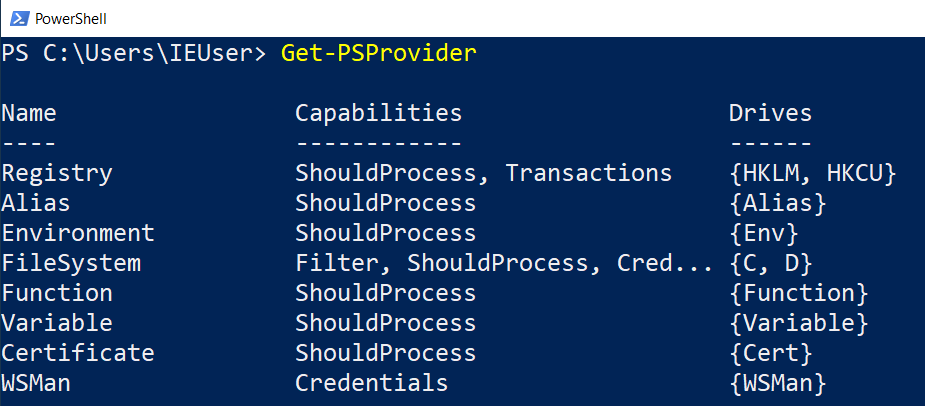
cat (Get-PSReadLineOption).HistorySavePath

Try setting some of the PSReadline options with the **Set-PSReadlineOption** command. If you set the save style to **SaveNothing**, do any of your commands show up in the history file?



# Providers

PowerShell uses **Providers** to make working with “*file-like*” systems convenient and consistent. View the providers with the **Get-PSProvider** function.



You can make use of **Providers** with functions like **Get-ChildItem**, **Set-ChildItem** and more. We are accustomed to using FileSystem providers such as the C:\ drive on a computer as shown below.

Text

Description automatically generated with low confidence

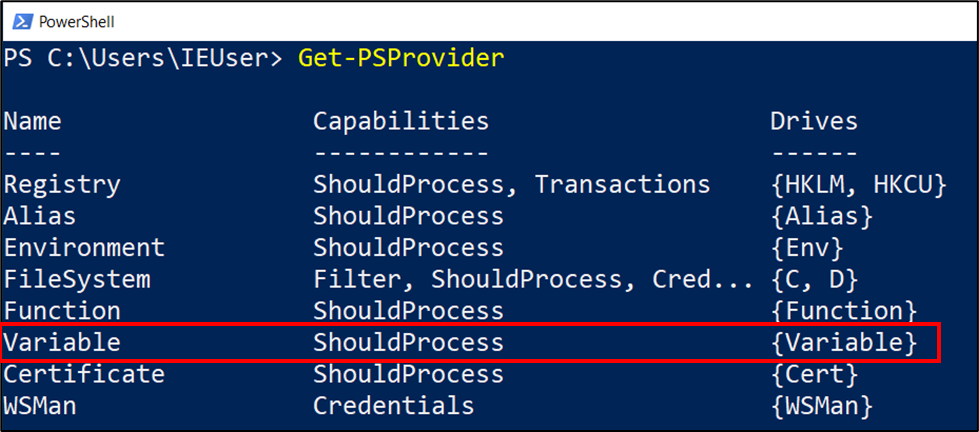
However, PowerShell has several more **Providers** for working with other similar data structures like the registry. We can read and write registry keys using the “Local Machine” registry provider **HKLM** or the “Current User” registry provider **HKCU** as shown below.

Graphical user interface, text

Description automatically generated

Above you can see how we used PowerShell’s registry provider to list Control Panel setting for the current user (HKCU). The image to the right shows the same registry setting displayed in the Registry Editor tool for comparison.

Did you notice the Variable provider?



View the content of the variable drive using the following command.

Get-ChildItem variable:

You will notice the output is the same as when we used the **Get-Variable** function. This is because the **Get-Variable** function simply prints the data in the variable drive.

Now let’s take a closer look at the **Alias** and **Env** providers/drives.

## Aliases

Aliases are shortcuts to functions. There are many built-in and you can also create custom ones. View the aliases in the current session with the **Get-Alias** function. Of course, you could also use **Get-ChildItem alias:** because it is one of PowerShell’s built-in Providers.

Timeline

Description automatically generated with low confidence

Here we see that we can use the **cls** or **clear** shortcuts to clear the screen or the **cat** shortcut to get-content from a file. This just saves us some typing and might match better with commands you already knew from other command shells.

Use the **Set-Ali**as function to create your own aliases. Here I create an alias for **Get-Date** so I can simply type **gd** to get the date.

Text

Description automatically generated

If you want to set an alias for something more complex, you can create an alias to a function as follows.

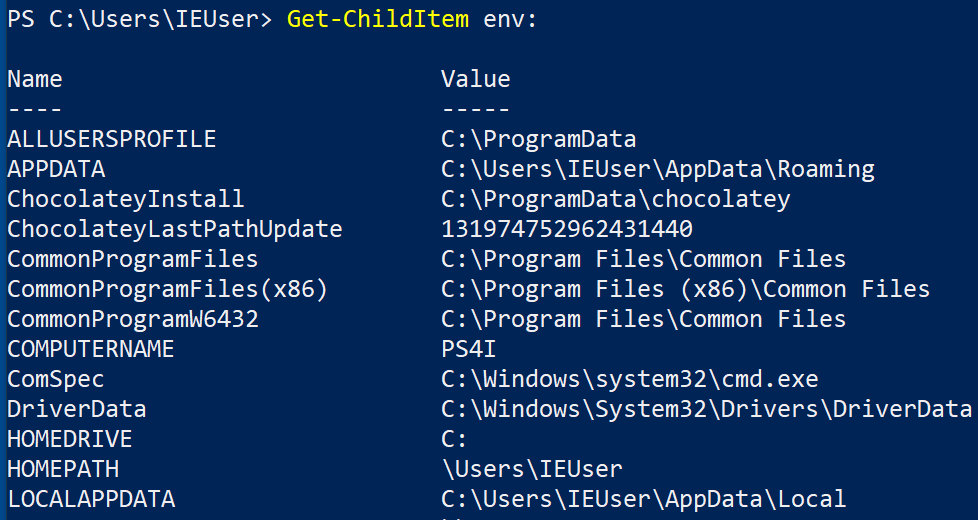
Graphical user interface, text

Description automatically generated

Note: Aliases only persist for the current session. If you want them to be available every time you use PowerShell, you will need to add them to your PowerShell profile which you will learn about soon.

## Environment Variables

The Environment Provider gives us the **env:** drive which we can use to view and set environment variables used by the operating system. How do you think we can view this drive? You guessed it …



One of the environment variables you will see used in PowerShell often is **TEMP**, which is the path to the temporary directory.

The **env** drive is implemented is such a way that you can refer to it using the following short hand. This makes it very convenient to use.

Text

Description automatically generated with medium confidence

# PowerShell Profile

The **profile** variable contains the path to a PowerShell script to be run every time a new PowerShell command shell is started. By default, the profile file does not exist, therefore nothing extra is executed when PowerShell starts. However, you can create this file and add things to it, such as your custom aliases that you want to always be available.

In the example below, we create a simple PowerShell profile to add our Get-Date alias, and then use it to print the date on the screen in any future PowerShell sessions that we start.

Graphical user interface, application, Word

Description automatically generated

Save your new profile in notepad and then start a new PowerShell session to see the effect.

Graphical user interface

Description automatically generated

Now that you’ve run several commands, experiment with the history feature by using the arrow keys to go back through your previously executed commands. Use **ctrl + R** and start typing part of a command you ran previously to find it in your history. Press **ctrl + R** again if the wrong version of the command you are looking for is matched.

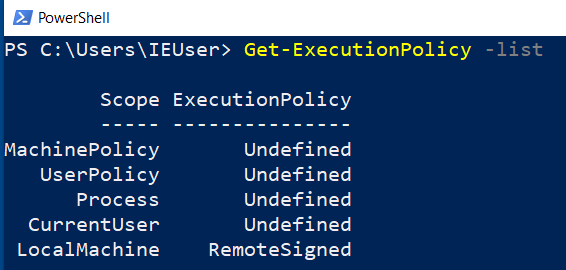
# Execution Policy

PowerShell utilizes an Execution Policy to help users avoid unintentionally running a script. The four policies are as follows:

* Bypass
* Restricted
* Unrestricted
* RemoteSigned
* AllSigned
* Default
* Undefined

The default execution policy for Windows 10 clients is **Restricted** but for some reason the Microsoft test VMs for our labs has it set to **RemoteSigned**.

Use the **Get-ExecutionPolicy** command to list the current policy settings.



Try running the **BasicScript** from the class samples directory. Then set the execution policy to block script execution. Finally, use the Bypass flag to subvert this protection mechanism on the fly.

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

In our lab environment, we don’t need to fight with the execution policy setting but here is the link to [15 ways to bypass PowerShell execution policy](https://www.netspi.com/blog/technical/network-penetration-testing/15-ways-to-bypass-the-powershell-execution-policy/) if you find yourself in need of that in the future. Remember that the execution policy should not be relied upon for blocking purposeful execution of malicious code.

This completes the PowerShell exploration lab. Hopefully you learned something new and are ready to learn some more about PowerShell in the coming lectures and labs.