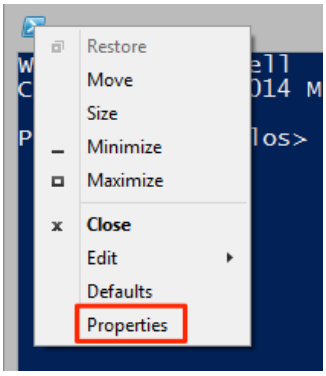
Microsoft PowerShell for Security Professionals

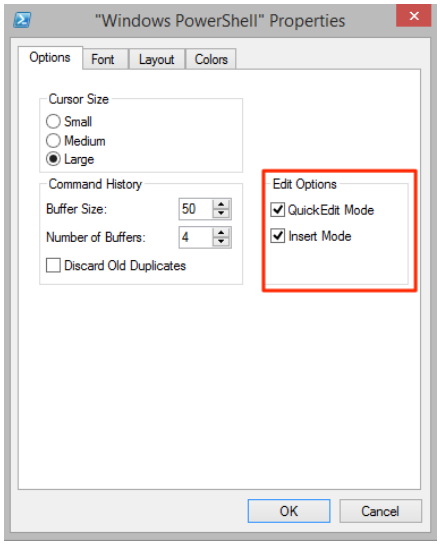
Lab Guide

Basics

Table of Contents

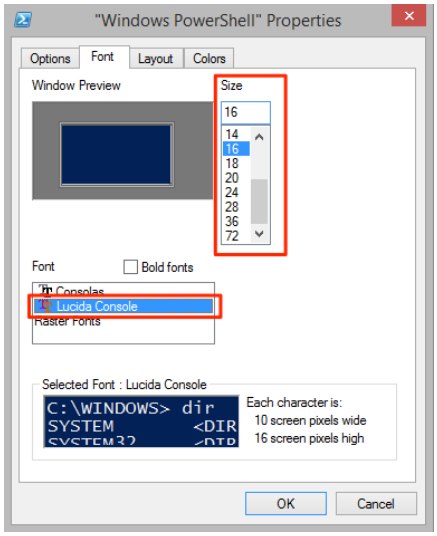
Setup the Console

1. On your Windows host open PowerShell.
2. If not pinned to the taskbar right click on the PowerShell icon on the taskbar and select **"Pin program to taskbar"**
3. **Right click** on the top corner of the opened PowerShell window where the PowerShell icon is shown and select Properties
   a. 

4. In the properties window on the **Options tab** check the boxes in **Edit Options** for **QuickEdit Mode** and **Insert Mode**
   a. 

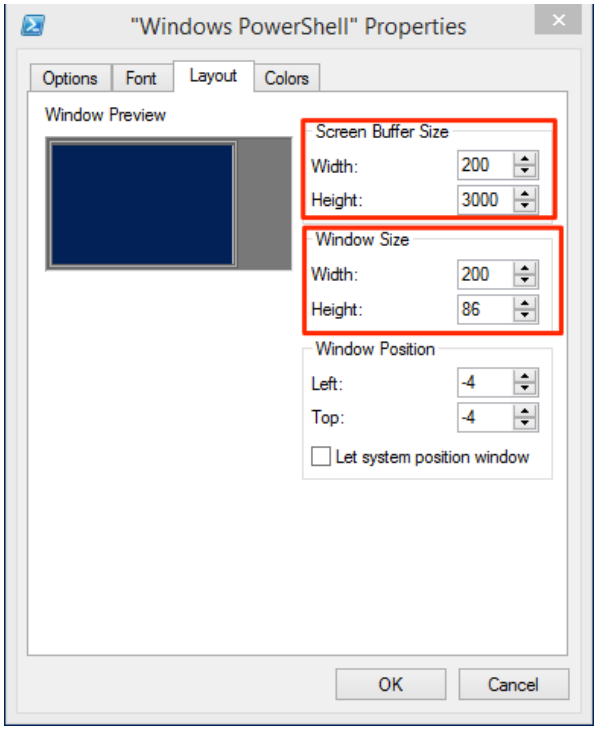
5. On the **Fonts tab** select **Lucida Console** and a **Font Size** of your liking for your console window.

a.



6. On the **Layout tab** specify a **width** you are conformable with in the **Screen Buffer** size and make sure it matches the value in the **Window Size width** field. Do make sure you do not have to scroll side to side to look at output. Specify a **Height** of 1000 or more in the **Screen Buffer Size** section.

a.


"Windows PowerShell" Properties
Options
Font
Layout
Colors
Window Preview
Screen Buffer Size
Width:
200
Height:
3000
Window Size
Width:
200
Height:
86
Window Position
Left:
-4
Top:
-4
Let system position window
OK
Cancel

Help

In this step, you will learn how to view the different levels of help content available for a cmdlet and basics of navigation of the help information.

1. At the Windows PowerShell command prompt, type the following command, and then press ENTER to see a list of available help topics.

```
help *
```

2. The command will fill an entire screen and then pause. Press **ENTER** to show the next output line, or press **SPACE BAR** to advance to the next page.
3. Press **SPACE BAR** until the command prompt returns. Alternatively, you can type the letter **Q** to cancel the output.
4. 5. To view help information about the Get-Command cmdlet, at the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
help Get-Command
```

5. To see detailed help for the Get-Command cmdlet, at the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
help Get-Command -Detailed
```

6. To view the entire help content for the Get-Command cmdlet, at the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
help Get-Command –Full
```

7. To view all examples and have them displayed in a window you can reference at the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
help Get-Command -ShowWindow
```

In the Window that is now opened with the help information click on **Settings** and leave only checked **Examples**.**Note: This lab will require Windows PowerShell 3.0 or higher,**

1. Using the Get-Help cmdlet in a PowerShell window running as Administrator search for a cmdlet to **schedule a job**.
2. Schedule a job that will run Update-Help -Force –Verbose every day at **6:00am**

Find the Right Command

In this step, you will list all available commands in Windows PowerShell.

1. To view the list of available commands, at the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
Get-Command
```

1. To view the list of available commands to with the noun **service** , at the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
Get-Command –Noun service
```

2. Look at the **source** field of the table shown and take note of the name show.
3. Use the Get-Help cmdlet to search for the word service and look at the fields shown. At the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
Get-Help service
```

2. To view all cmdlets in the module that contains the service management cmdlets, at the Windows PowerShell command prompt, type the following command, and then press **ENTER**.

```
Get-Command –Module Microsoft.PowerShell.Management
```

In this step, you will practice finding the appropriate command and what parameter to use in Windows PowerShell.

1. Using **Get-Command** find all commands related to working with an **Alias**.
2. Using help find out how to get all aliases associated with **Get-ChildItem**.
3. With the alias commands found create a new alias of **'ll' (Double lowercase L)** for **Get-ChildItem**.

Providers

In this step, you will list the available providers and the drives, which make use of these providers. You will also create a new drive using the registry provider.

1. To display a list of the available providers, type the following command, and then press ENTER.

```
Get-PSProvider
```

2. To display a list of the available drives, type the following command, and then press ENTER.

```
Get-PSDrive
```

3. To create a new drive for the HKEY_CLASSES_ROOT hive in the registry using the Registry provider, type the following command, and then press ENTER.

```
New-PSDrive –Name HKCS –PSProvider Registry –Root "HKEY_CLASSES_ROOT"
```

4. To browse to the newly created drive, called HKCS, as if you were working with a drive in the file system, type the following commands, pressing ENTER after each one.

```
cd HKCS:
dir .ht*
```

5. To change the current folder to another folder inside the HKCS drive, and then list its contents, type the following commands, pressing ENTER after each line.

```
cd .hta
```

```
dir
```

File System

- Open a PowerShell window as administrator.
- List all providers and check their capabilities.
    - Using Get-Help cmdlet get the help for New-Item and in the **–Path** parameter specify **C:** and then **WSMan:\localhost\ClientCertificate**
    - Did the Synopsis of the cmdlet change?
    - List all parameter but in path specify **WSMan:\localhost\Plugins**.
    - Did the parameter list change?

Registry

In this lab we will learn the basics of working with the registry.

- Open a PowerShell window.
    - Using help look for cmdlets that allow enabling a **transaction**.
    - Navigate to **HKCU:** and **start a transaction**.
    - In **HKCU:** create a Item called **_ClassKey** and make use a transaction.
    - Under **_ClassKey** create another one call **PSSEC** and make use a transaction.
    - Set a String property on PSSEC call "**Moto**" with a value **"Blue is the new Black"** and make use a transaction.
    - Check the property on PSSEC key using Regedit.
        - Is the key there?
        - Complete the transaction, refresh the view in regedit and check if it is there.
    - Delete **__ClassKey** from PowerShell**.**

Extending the Shell Modules

Install modules

In this lab we will install the PowerShell modules provided in the class and practice loading the module and listing its command.

1. Change the execution policy on the system from a PowerShell window running as Administrator use the **Set-ExecutionPolicy** to **RemoteSigned**.
2. If a WindowsPowerShell folder does not exist in the users Documents folder, using PowerShell create a folder with the following path:
    a. ***"$($env:USERPROFILE)\Documents\WindowsPowerShell\Modules"***
3. Copy in the to the Modules folder all modules included in the Modules folder provided for the class. Make sure each module is individually under the Modules folder.
4. Open a new Windows PowerShell window and list the modules available. Ensure the modules recently copies are shown.
5. Load the Posh-SSH module using the verbose flag.
6. Using Get-Command cmdlet list only the commands in the module.

Create a New Module

In this lab we will create a new module that we will use during class.

1. In **"$($env:USERPROFILE)\Documents\WindowsPowerShell\Modules"** create a folder named **Posh-SecClass usingt** the **New-Item** cmdlet
2. Create an empty psm1 file named **Posh-SecClass.psm1** inside the folder with the **New-Item** cmdlet.
3. Create a new module manifest using **the New-ModuleManifest** cmdlet. It should have the following properties:
    a. Name for manifest: **Posh-SecClass.psd1**
    b. Author : **Your name**
    c. Root Module: **Posh-SecClass.psm1**
    d. Version: **0.1**
4. Open the created module manifest in ISE and explore the options that are commented and set.

~~Conflicting Cmdlets and Functions~~

Pipeline

Filtering with Where-Object

1. Cast the content of the Nessus report as a XML object in to a variable, reading the content of the file using **Get-Content** cmdlet, type the following commands, press ENTER

```
[xml]$report = Get-Content -Raw .\DevLabRepor.nessus
```

2. Open the report in ISE and navigate the properties of the XML object in PowerShell to see how the XML structure is represented and objectified by PowerShell.

3. Explore the variable and how the XML object makes navigating the structure via properties so easily, type the following commands, press ENTER

```
$report
$report.NessusClientData_v2
$report.NessusClientData_v2.Report
$report.NessusClientData_v2.Report.ReportHost
$report.NessusClientData_v2.Report.ReportHost[0].ReportItem
```

4. As we can see, each Host in the scan can be found under **ReportHost**. To process host by host we will have to use the **Foreach-Object** cmdlet, type the following commands, press ENTER

```
$report.NessusClientData_v2.Report.ReportHost | ForEach-Object
{$_.HostProperties.tag; "`n`n"}
```

5. Search for report items that are for findings in port 22, , type the following commands, press ENTER

```
$reporhosts = $report.NessusClientData_v2.Report.ReportHost

$reporhosts | ForEach-Object {$_.ReportItem} | Where-Object {$_.port -
eq 22}
```

6. For each host iterate through their **ReportItems** ,filter out only those that have the property **exploit_available** set to **$true** and pipe the results to **Out-GridView**
7. For each host iterate through their **ReportItems** ,filter out only those that have the property **exploit_available** set to **$true,** a **severity** of **3** or more and pipe the results to **Out-GridView**

Compare Objects

1. Using **Get-Service** cmdlet list all services on your host and pipe the output to **Export-CliXML** file and name the file **base.xml**
2. Check the status of the **BITS** service and change it.
3. Using **Get-Service** cmdlet list all services on your host and pipe the output to **Export-CliXML** file and name the file **current.xml**
4. Using **Import-CliXML** cmdlet import the XML files in to 2 different variables.
5. Use **Compare-Object** cmdlet to compare both sets specifying the properties **Status** and **Name**.

Renaming Properties

This lab will cover modifying the names of

1. Import the CSV file included in lab materials named hosts.csv using **Import-CSV** cmdlet and look at the names of the properties.
2. Using **Get-Help** cmdlet look at the properties of the **Test-Connection** cmdlet (PowerShell version of Ping), identify what parameter represents what we want to ping and if it accepts values from the pipeline and how.
3. Using **Get-Help** on **Select-Object** find a reference example that shows how would you change the name of the **IP** property to **ComputerName** and pipe it to **Tes-Connection** sending only one message.

Pipeline Processing

ByValue Lab

In this lab, we will create an advanced function that can take any object type from the pipeline and process so as to show how an advanced function manages the pipeline.

1. Using PowerShell ISE press **Ctrl-J** and select the snippet for **Cmdlet (advanced function).**
2. Change the verb to **Get** and noun to **Pipeline.**
3. Create 1 parameters with the following parameters
    a. Name: **PipelineValue**
    b. Mandatory: **True**
    c. ValueFromPipeline: **True**
    d. Position: **0**

```
# Value frome Pipeline
[Parameter(Mandatory=$true,
           ValueFromPipeline=$true,
           Position=0)]
$PipelineValue
```

4. The function will have 3 code blocks in it (**Begin**, **Process** and **End**) in the Begin process block save in to a variable a text message that says "I only run once" and on the next line just type the variable name.
5. On the Process block save in to a variable the string with the interpolation of the parameter as "I got in the pipeline $($PipelineValue)" and then in the next line type only the variable name so it is returned when executed.
6. On the Begin process block save in to a variable a text message that says "I run last" and on the next line just type the variable name.
7. Save the file and reload the module using **Import-Module** cmdlet with the **Force** and **Verbose** parameters.
8. Use **Get-Help** cmdlet to look at the function you just create. Run the cmdlet with the **Full** parameter and look at the parameter for the function and its option making sure it accepts pipeline input and the input is by value.

9. Create a range of numbers using the expression **(1..10)** and pipe the results in to the function you just created. Each number should be processed by the Process code block confirming the function was written correctly.

Static Type ByValue

In this lab, we will look at how to specify a specific object type for a parameter and how it behaves in the pipeline.

1. Edit the advanced function created in the previous lab and set the parameter to be statically typed one to be a 32bit integer.

```
# Value frome Pipeline
[Parameter(Mandatory=$true,
           ValueFromPipeline=$true,
           Position=0)]
[int32]$PipelineValue
```

2. Create a range of numbers and using ForEach-Object cmdlet and string interpolation turn each object in to a string piping each to the function

```
(1..10) |foreach-object {"$($_)"} | Get-Pipeline
```

3. PowerShell default behavior of trying to alter types when possible to match the receiver should have permitted this to run by turning the string in to integer.
4. Execute the same command but this time the interpolation string add the word “User” so it will generate strings like User1, User2.. This should generate an error that it could not match the type.

Create Custom Object

In this lab, we will create PSCustom objects with a set property we will use with another function that will take from the pipeline ValueFromPipelineByPropertyName

1. Using PowerShell ISE press Ctrl-J and select the snippet for Cmdlet (advanced function).

2. Change the verb to **New** and noun to **TestObjects**.
3. Remove all parameters from the function.
1. In the Process block type the following code:

```
Process
{
    (1..10) |
        ForEach-Object {
            New-Object -TypeName psobject -Property @{
                value = $_
                Type = 'Integer'
            }
        }
}
```

2. Save the file and reload the module using Import-Module cmdlet with the Force and Verbose parameters.

3. Run the advance function to see the objects it returns. Run the function again and pipe the results to Get-Member to see the type and properties.

4. Using PowerShell ISE press Ctrl-J and select the snippet for Cmdlet (advanced function).

5. Change the verb to Get and noun to Pipeline.
6. Create 1 parameters with the following parameters

    - Mandatory: True
    - ValueFromPipelineByPropertyName: True
    - Position: 0
    - Static type: Int32

```
# Integer value to process.
[Parameter(Mandatory=$true,
           ValueFromPipelineByPropertyName=$true,
           Position=0)]
[int32]$value
```

The Begin, Process and End blocks are the same as those of the Get-Pipeline Function.

```
Begin
{
    $message = 'I only run once'
    $message
}
Process
{
    $message = "I got in the pipeline $($value)"
    $message
}
End
{
    $message = 'I run last'
    $message
}
```