PowerShell Scripting for Kids

SECURITY B-SIDES DC 2017

ANDREW COLE

Introductions

Get-Instructor

- My name is Andrew Cole.
- I'm a Computer Network Exploitation (CNE) instructor for the Department of Defense.
- Before that, I was a Soldier in the US Army, where I was a Cryptologic Network Warfare Specialist.
- I have three kids, two daughters and a son.
- I LOVE POWERSHELL!!!!

Get-Students

- Now, who are you?
- What is your computer experience?
- Have you ever tried coding before?

What is PowerShell?

- Is it –
  - A way to send commands to a computer?
  - A tool to manage a computer?
  - A tool to manage a network of computers?
  - A scripting language?
- …YES!! All of the above!
  - PowerShell can do all of these things, and much, much more.
- For the purposes of this class, we will be focusing on the scripting component of PowerShell.

What is PowerShell?

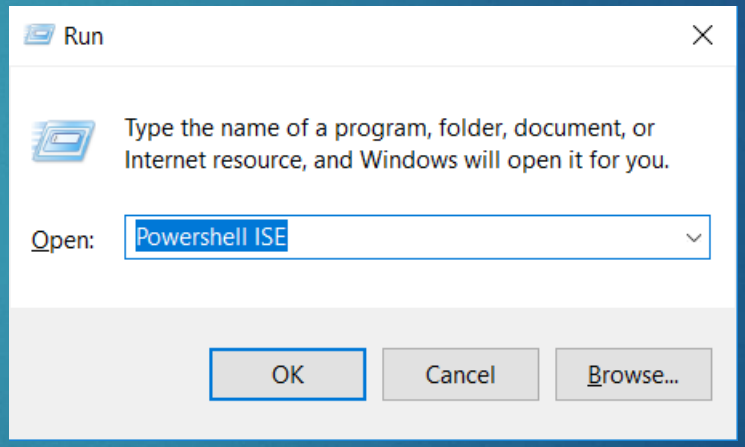
- Unlike most scripting languages, PowerShell is ***'object oriented'***.
  - What does that mean?
- Instead of parsing text, it parses objects instead.
  - But what is an object?
- That's a little more complicated…
  - To explain that, we'll have to talk about classes, instances, properties, and methods.

Object Oriented Terms

- Classes are groups of objects which have a lot of things in common.
  - We'll use the 'Dog' class.
- Instances are examples of a class.
  - Rex and Fido are instances of the dog class.
- Properties are things that make the instances what they are.
  - Rex has 4 legs, fur, and a tail.
- Methods are things you can do with an instance.
  - Walk it, pet it, or play fetch with it.
- An object is just an instance that is currently created on the computer.

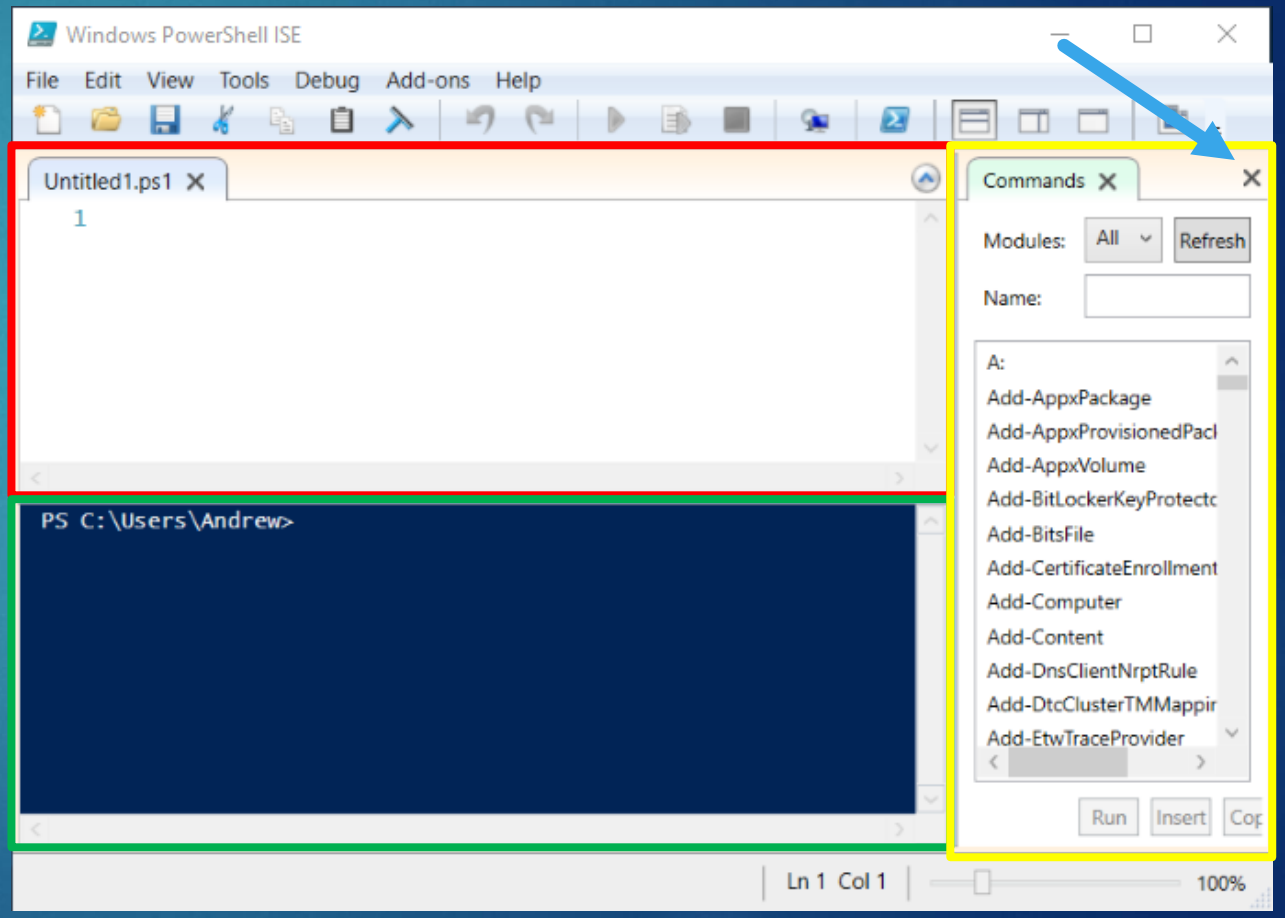
PowerShell ISE

- The program we will be using to write our scripts in the PowerShell Integrated Scripting Environment.
- Usually it is just called the PowerShell ISE.
- To open it, just press the ‘windows’ key and the ‘r’ key, then type ‘PowerShell ISE’ in the window that pops up.



PowerShell ISE

- The ISE has three separate panes, or areas:
  - #1 is the scripting pane
    - It's where we will write our scripts.
  - #2 is the console pane
    - It's where we will test out commands
  - #3 is the module/add-on pane
    - We won't be using this, you can click the 'X' to close it.



Execution Policy

- By default, scripts are not allowed to be run in PowerShell.
  - Which doesn't work well for a scripting class…
- We will need to enable this feature.
- To do so, right-click on the PowerShell (or PowerShell ISE) icon in your taskbar and click 'Run As Administrator'.
- In this new PowerShell, type the following:

```
PS C:\> Set-ExecutionPolicy RemoteSigned
```

- If it asks if your sure, type Y, then close the elevated PowerShell.

Writing Scripts

- Scripting process
  - Think of what you want to do
  - Write pseudocode (comments work great for this)
  - Write a command & test it
  - Put the commands in a function/script & test it
  - Have someone else look over you code & test it
  - Publish & share code

- In this class, all the pseudocode will already be written for you. ☺

Variables & User Interaction

SECTION 1

Variables

- The most important part of any scripting language is the ability to create variables.
- Variables are containers in memory that can hold objects (or anything else for that matter).
- They are called variables because what is inside them can vary, or change.
- They allow us to save the results of a command for further use, without having to run the command again.

Variables

- In PowerShell, variables are easy to spot because the start with a '$', and there are a couple different ways to create them.

- The easiest way is to just declare them. Let's say you wanted to save the phrase 'PowerShell is cool.' to a variable called 'phrase'. To do so you would type the following:

```
PS C:\> $Phrase = 'PowerShell is cool.'
```

- Now, whenever you wanted to print 'PowerShell' to the screen, just call the variable by typing its name.

```
PS C:\> $Phrase
PowerShell is cool.
```

Variables

- Variables can hold more than just a single string of text. Try typing the following:

```
PS C:\> $Procs = Get-Process
PS C:\> $Procs
```

  - What did it return?
  - A listing of all the processes (programs) running on your computer.

- Almost every script uses variables for something. Uses include:
  - To save input from the user
  - To compare two or more objects
  - To see if something exists or not

User Interaction

- One of the easiest uses for variable it to interact with the user.
  - We can ask the user a question, and save their answer to a variable.
  - We can then use their answer in our response back to them.

- To do this, we will use two cmdlets (pronounced ‘commandlets’, a fancy word for PowerShell commands):
  - Read-Host
    - Collects information from the user
  - Write-Output
    - Displays information to the user

User Interaction

- Read-Host writes a line of text to the screen and ‘reads’ whatever the user responds.

```
PS C:\> Read-Host ‘What is your name?’
What is your name?:  Andrew
Andrew
```

- Typically this is saved as a variable, as below:

```
PS C:\> $name = Read-Host ‘What is your name?’
What is your name?:  Andrew
PS C:\> $name
Andrew
```

User Interaction

- Write-Output simply writes a line of text, the contents of a variable, or a combination of both to the screen.

```
PS C:\> Write-Output 'PowerShell is Awesome!'
PowerShell is Awesome!
PS C:\> $20 = 'twenty'
PS C:\> Write-Output "The number 20 is spelled $20."
The number 20 is spelled twenty.
```

- Note that the first example above uses single quotes, while the second uses double quotes. These are very different.
  - ' ' prints as a string ($20 would be displayed as $20)
  - " " will resolve any variables ($20 would be displayed as twenty)

Creating Functions

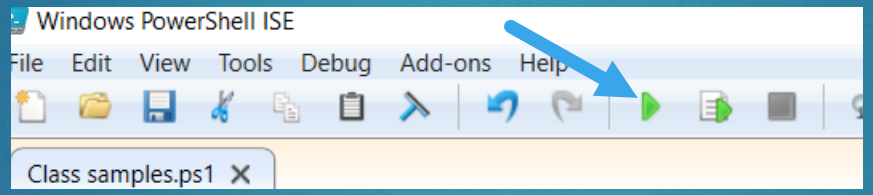
- A series of commands which are grouped together into a single command is called a function.

- Functions are created as indicated below:

```
Function <name goes here>
{
    <command 1>
    <command 2>
    etc…
}
```

Creating Functions

- Once the function is created, it must be imported before it can be called (run). There are two ways to do this:
  - Open the script in the ISE and press the play button



  - Save the script to a file and import it.

```
PS C:\> Import-Module <path to file>
```

- Once the function is loaded into memory, it can be run by typing the function name.

Example 1 & Project 1

MAD LIBS

Arrays & Randomization

SECTION 2

Arrays

- Now that we understand how variables work, we can move on to arrays. An array is just a variable that holds more than one thing at a time.
- If we think of a variable as a box that holds something, an array would be like an egg carton, with separate locations which each hold a different object.
- Arrays are very useful for dealing with large groups of objects.

Arrays

- Let's create an array of colors called $egg_carton.
- Arrays are defined just like variables, except the definition begins with @(and ends with).

```
$egg_carton = @(
    'green'
    'pink'
    'blue'
    'yellow'
    'green'
    'pink'
    'yellow'
    etc…     )
```

Arrays

- In our array, called $egg_carton, we can also number each spot in the carton (just remember that computers start counting at 0, not 1). This number is called the index number.
- Objects can be called by their index number. This is done in the following format:
  - $arrayname[indexnumber]
- What color would you say was at $egg_carton[5]?

Get-Random

- Usually, we want scripts to do the exact same thing every time they are run.
  - But not always…
- Sometimes we want a different outcome each time.
  - Like if you wrote a script that simulated rolling dice. It would be a really bad program if it rolled the name number every time…
- To get this possibility of different outcomes (known as randomization) we will use the Get-Random cmdlet.

Get-Random

- The Get-Random cmdlet takes two parameters (options)
  - Minimum (the lowest number it can choose)
  - Maximum (the highest number it can choose)

- There is a small issue with the Get-Random cmdlet, it will never pick the highest number in the range.
  - If you wanted a number between 0-10, the maximum would have to be set to 11.
    - If it was set to 10 it would only pick numbers 0-9

```
PS C:\> Get-Random –Minimum 0 –Maximum 11
```

Random Array Object

- We can combine Get-Random with an array to select a random object. Let's say you were getting a pet, but couldn't decide between a dog, a cat, and a rabbit. We could put all three in an array, and use Get-Random to select one for you, as below:

```
PS C:\> $pets = @( 'Dog', 'Cat', 'Rabbit')
PS C:\> $num = Get-Random –Minimum 0 –Maximum 3
PS C:\> $selection = $pets[$num]
PS C:\> Write-Output "You should get a $selection"
```

  - If the number generated was 0, it would have selected a dog.
  - If the number generated was 1, it would have selected a cat.
  - If the number generated was 2, it would have selected a rabbit.

Example 2 & Project 2

MAGIC 8 BALL

Loops

SECTION 3

Loops

- Another key concept in scripting is the loop. A loop just repeats an action, just like a race car driving around a loop.
- Some loops go on forever (or at least until they are manually stopped by a user).
- Some loops go on until a set condition is met.
- The conditions for the loop are enclosed in ( ).
- The action to take it the conditions are met are enclosed in { }.

Loops

- To make an infinite loop, (one that goes on forever), set the condition to $true.
  - $true is a constant variable.
  - Since $true will always be equal to true, the condition will always be met, and the loop will go on forever.
- Put the action you want to repeat inside the { }.
  - In this example, it is write-output "FOREVER!"

```
While($true)
{
    Write-Output "FOREVER!"
}
```

```
FOREVER!
FOREVER!
FOREVER!
FOREVER!
FOREVER!
FOREVER!
FOREVER!
FOREVER!
FOREVER!
FOREVER!
(you get the idea…)
```

Loops

- To make a conditional loop, define the condition to be met between the ( ).
  - In this case, the loop will repeat as long as $x is less than (indicated by -lt) 3.
- Put the action you want to repeat inside the { }.
  - In this example, it will display the value of the variable $x, and then add 1 to it.
    - Another way to write '$x = $x + 1' is '$x += 1'

```
$x = 0
While($x –lt '3')
{
    Write-Output "X is equal to $x"
    $x = $x + 1
}
Write-Output "Loop ended."
```

```
X is equal to 0
X is equal to 1
X is equal to 2
Loop ended.
```

Comparison Operators

- That last loop uses something called a comparison operator.
  - A comparison operator compares to things. For example:
    - Apple and Apple are the same, apple and orange are not.
  - The commonly used comparison operators are:
    - -eq (equals)
    - -ne (not equal to)
    - -lt (less than)
    - -gt (greater than)
    - -le (less than or equal to)
    - -ge (greater than or equal to)

Example 3 & Project 3

KNOCK, KNOCK JOKE

Conditional Statements

SECTION 4

Conditional Statements

- The last scripting concept we are going to cover is conditional statements.
- Conditional statements often use comparison operators like loops, but the don't repeat commands. Instead the determine what commands are run and which aren't.
- There are three conditional statements we will focus on:
  - If statements
  - If/else statements
  - If/elseif/else statements

If Statements

- If statements are fairly simple. If the condition is met, do the action.
- For example, if you finish your dinner you can have dessert.

```
If($dinnerfinished –eq $true)
{
    Get-Dessert
}
```

  - If you don't' meet the condition (you didn't finish your dinner), then the action is not completed (you don't get dessert ☹)

If Statements

- Consider the example below.
  - If the variable $number is greater than or equal to 10, then the action will occur, and write "Two digit number" to the screen.
  - Since $number is set to 9, which is not less than or equal to 10, nothing happens.

```
$number = 9
If($number –ge 10)
{
    Write-Output "Two digit number"
}
```

- In this example below, however, $number is set to 15, which IS greater than or equal to 10.
  - Because of this the action executes, and the statement is printed to the screen.

```
$number = 15
If($number –ge 10)
{
    Write-Output "Two digit number"
}
```

```
Two digit number
```

If/Else Statements

- If/else statements say if the first condition is met, do the first action. Otherwise, do the second action.

- Think of it like flipping a coin.
  - If you get a new videogame, and you and a sibling both want to play it, you might flip a coin to see who goes first.
    - Let's say you called heads.
  - ***IF*** the coin lands on heads, you get to play first, or
  - ***ELSE*** (meaning it did not land on heads), your sibling gets to play first.

If/Else Statements

- Consider the examples below.
  - If the variable $score is equal to $true, it will write GOOOOAAALL!!
  - Otherwise, it will write Just missed!

```
$score = $true
If($score –eq $true)
{
    Write-Output “GOOOAAALL!!”
}
Else
{
    Write-output “Just missed!”
}
```

GOOOAAALL!!

```
$score = $false
If($score –eq $true)
{
    Write-Output “GOOOAAALL!!”
}
Else
{
    Write-output “Just missed!”
}
```

Just missed!

If/Elseif/Else Statements

- If/Elseif/Else statements are just like if/else, except there is an extra conditional statement (sometimes several) called ***elseif*** between the ***if*** and the ***else***.
- Think of it like you are going to a toy store, and they have two toys you want:
  - A lego set for $10
  - A fidget spinner for $5
- Depending how much money you have, you can get both of them, one of them, or neither of them.

If/Elseif/Else Statements

- In other words:
  - IF you have $15 or more, you get both. Or
  - ELSEIF you have more than $10 but less than $15 you get the lego set. Or
  - ELSEIF you have more than $5 but less than $10 you get the fidget spinner. Or
  - ELSE you don't get anything.
    - Because none of the first 3 conditions were met.

```
If($money –ge 15)
{
    $toy = legos_and_spinner
}
Elseif($money –ge 10)
{
    $toy = legos
}
Elseif($money –ge 5)
{
    $toy = spinner
}
Else
{
    $toy = $null
}
```

Example 4
& Project 4

HI - LO

Keep on PowerShelling!!!