A layman's guide to PowerShell 2.0 remoting

Ravikanth Chaganti

Learn the basics of PowerShell 2.0 remoting, methods of remoting and how to use remoting to manage systems in a datacenter.

A layman's guide to PowerShell 2.0 remoting

Ravikanth Chaganti

Jan Egil Ring

Acknowledgments

Thanks to everyone who read my blog posts on PS remoting and provided feedback. Your encouragement and support helped me write quite a bit about remoting and now this e-book. Thanks to Jan Egil Ring (http://blog.powershell.no/) for contributing to appendix B in this updated version of the ebook.

Contents

Part1

Chapter 1: Introduction to remoting

Traditional remoting in PowerShell

A few cmdlets in PowerShell support accessing information on a remote system. These cmdlets have a –ComputerName parameter. For example the following cmdlets support the computername parameter and hence can be used to access information from a remote computer.

- Get-WmiObject
- Invoke-WmiMethod
- Limit-EventLog
- Set-Service
- Set-WmiInstance
- Show-EventLog
- Stop-Computer
- Clear-EventLog
- Get-Counter
- New-EventLog
- Register-WmiEvent
- Remove-EventLog
- Remove-WmiObject
- Restart-Computer
- Get-EventLog
- Get-HotFix
- Get-Process
- Get-Service
- Get-WinEvent

The remoting capability of these cmdlets is independent of PowerShell. It is up to the cmdlet author to implement the remote access using methods such as remote procedure call (RPC), etc. This method of remoting can be called traditional remoting or classic remoting.

One obvious disadvantage is that not all PowerShell cmdlets implement this type of remoting. So, for example, if you want to execute Get-PSDrive or Get-ChildItem remotely on a different computer, it is not possible. This is where the new PowerShell 2.0 remoting feature plays an important role. So, throughout this guide, whenever we refer to remoting, we refer to the new remoting technology but not traditional or classic remoting methods.

Overview of PowerShell 2.0 remoting

One of the most exciting and important features of PowerShell 2.0 is the remoting capability. PowerShell remoting enables management of computers from a remote location. Remoting is built on top of Windows remote management (WinRM)[1]. WinRM is Microsoft's implementation of WS-Management[2] protocol.

This feature enables what is known as Universal Code Execution Model[3] in Windows PowerShell 2.0. UCEM means that whatever runs locally should run anywhere. PowerShell remoting also lets you import remote commands in to a local session — a feature known as **implicit remoting** and also enables you to save or export these imported commands to local disk as a module for later use. There are bunch of other features such as interactive sessions, etc. We will look in to all these features -- one thing at a time.

PowerShell remoting allows for multiple ways of connecting. These ways include interactive (1:1), fan-out (1: many), and fan-in (many: 1 by using the IIS hosting model, for example, Quest Software's MobileShell[4]). This guide will walk though each of these ways and explain how to configure your system for these scenarios.

PowerShell 2.0 remoting requirements

To enable PowerShell remoting, all computers participating in remote management should have the following software

1. Windows PowerShell 2.0
2. NET framework 2.0 SP1 or later
3. Windows Remote Management (WinRM) 2.0

All of the above are installed by default on Windows 7 and Windows Server 2008 R2. However, earlier versions of Windows will require you to download the updates from Microsoft website and install them yourself.

PowerShell 2.0 and WinRM 2.0 are included as a part of Windows Management Framework download and are available for Windows XP, Windows Server 2003, Windows Vista and Windows Server 2008. WinRM 2.0 and PowerShell 2.0 can be installed on the following supported operating systems

1. Windows Server 2008 with Service Pack 1
2. Windows Server 2008 with Service Pack 2
3. Windows Server 2003 with Service Pack 2
4. Windows Vista with Service Pack 2
5. Windows Vista with Service Pack 1
6. Windows XP with Service Pack 3

[1] WinRM: http://msdn.microsoft.com/en-us/library/aa384426(VS.85).aspx
[2] WS-Management: http://msdn.microsoft.com/en-us/library/aa384470(VS.85).aspx
[3] UCEM as explained by Jeffery Snover: Universal Code Execution Model
[4] MobileShell

7. Windows Embedded POSReady 2009
8. Windows Embedded for Point of Service 1.1

PowerShell 2.0 remoting is supported only on the operating systems listed above.

To be able run scripts and commands on remote computers, the user performing remote script execution must be

- a member of the administrators group on the remote machine **OR**
- should be able to provide administrator credentials at the time of remote execution **OR**
- should have access the PS session configuration on the remote system

For a complete discussion on PS Session configurations refer to chapter << >>.

Also, on client OS versions of Windows such as Windows Vista and Windows 7, network location must be set either to Home or Work. WS-Management may not function properly if the network location for any of the network adapters is set to **public**.

Overview of remoting cmdlets

This section provides a quick overview of some of the important cmdlets that are used in PowerShell remoting. This list will also include cmdlets that are not directly used within remoting but help configure various aspects of remoting. The knowledge of some of these cmdlets such as WSMan cmdlets is not mandatory for basic usage of PowerShell remoting. Subsequent chapters will discuss these cmdlets in detail.

Enable-PSRemoting

The Enable-PSRemoting cmdlet configures the computer to receive Windows PowerShell remote commands that are sent by using the WS-Management technology. This cmdlet will be the first one to run if you want to use PowerShell 2.0 remoting features and needs to be run just once. This cmdlet internally calls Set-WSManQuickConfig to configure WinRM service, enable firewall exceptions for WS Management and finally enables all registered PowerShell configurations.

Note: You need to enable PowerShell remoting only if you want the computer receive commands from a remote machine. To only send commands to a remote machine, you don't need to enable PowerShell remoting.

Disable-PSRemoting

The Disable-PSRemoting cmdlet disables all PowerShell session configurations on the local computer to prevent the computer from receiving any remote commands. You will have to manually stop the WinRM service if you don't want the service to be running after you disable PowerShell remoting.

Invoke-Command

The Invoke-Command cmdlet runs commands on a local or remote computer and returns all output from the commands, including errors. With a single Invoke-Command command, you can run commands

on multiple computers. This cmdlet — in its default form — opens a session for running a command against a remote computer and closes it once the execution is complete. This method may be slow and can be worked around by specifying pre-defined session information.

New-PSSession

Invoke-Command cmdlet supports specifying an existing session to enhance the speed of overall command execution. By specifying an existing session, we eliminate the need for creating/destroying the sessions on the fly. New-PSSession cmdlet can be used to create a persistent connection to a remote computer. By creating a persistent session, we will be able to share data, such as a function or the value of a variable between different commands executing within the PSSession.

Enter-PSSession

Enter-PSSession cmdlet starts an interactive session with a single remote computer. During the session, the commands that you type run on the remote computer, just as though you were typing directly on the remote computer. You can have only one interactive session at a time. You can specify the PSSession you created using New-PSSession as a parameter to this cmdlet.

Exit-PSSession

Exit-PSSession exits an interactive PS Session created using Enter-PSSession cmdlet.

Get-PSSession

The Get-PSSession cmdlet gets the Windows PowerShell sessions (PSSessions) that were created in the current session. This cmdlet gets all the PSSessions returns all the PSSessions in to a variable when no parameters are specified. You can then use the session information with other cmdlets such as Invoke-Command, Enter-PSSession, Remove-PSSession, etc.

Remove-PSSession

The Remove-PSSession cmdlet closes PS session(s). It stops any commands that are running in the PSSessions, ends the PSSession, and releases the resources that the PSSession was using. If the PSSession is connected to a remote computer, Remove-PSSession also closes the connection between the local and remote computers.

Import-PSSession

Import-PSSession cmdlet uses the implicit remoting feature of PowerShell 2.0. Implicit remoting enables you to import commands from a local/remote computer in to an existing PS session and run those commands as if they were local to the session.

Export-PSSession

The Export-PSSession cmdlet gets cmdlets, functions, aliases, and other command types from another PSSession on a local or remote computer and saves them to local disk as a Windows PowerShell module. We can now use the Import-Module cmdlet to add the commands from the saved module to a PS Session.

Register-PSSessionConfiguration

Any PS session created using Invoke-Command or New-PSSession or any other PowerShell remoting cmdlet for that matter uses the default PS Session configuration as specified in the $PSSessionConfigurationName variable. PS Session configuration determines which commands are available in the session, and it can include settings that protect the computer, such as those that limit the amount of data that the session can receive remotely in a single object or command. So, you can use the Register-PSSessionConfiguration cmdlet creates and registers a new session configuration on the local computer.

Unregister-PSSessionConfiguration

The Unregister-PSSessionConfiguration cmdlet deletes registered session configurations from the computer. It is possible to delete the default PSSession configurations (Microsoft.PowerShell or Microsoft.PowerShell32) using this cmdlet. In such a case, you can use Enable-PSRemoting cmdlet to re-create and register the default PS Session configurations.

Disable-PSSessionConfiguration

Disable-PSSessionConfiguration disables a registered PS Session configuration. Remember, this only disables the configuration but not un-register or delete the information from local computer. These disabled session configurations cannot be used to establish a remoting session.

Enable-PSSessionConfiguration

The Enable-PSSessionConfiguration cmdlet re-enables registered session configurations that have been disabled by using the Disable-PSSessionConfiguration cmdlet.

Get-PSSessionConfiguration

The Get-PSSessionConfiguration cmdlet gets the session configurations that have been registered on the local computer.

Set-PSSessionConfiguration

The Set-PSSessionConfiguration cmdlet changes the properties of the registered session configurations on the local computer.

Test-WSMan

PowerShell remoting requires WinRM service to be running on the remote machines. You can use Test-WSMan cmdlet to quickly check if you can establish a remoting session with other computers. If WinRM is not enabled on remote machine, you can safely assume that PowerShell remoting is not enabled. However, you cannot assume that PowerShell remoting is enabled just by verifying that WinRM service is running. Remember, this cmdlet checks only for WinRM service and remoting requires many other components to function.

Enable-WSManCredSSP

PowerShell remoting supports CredSSP authentication and the same can be enabled by using Enable-WSManCredSSP cmdlet. The Enable-WSManCredSSP cmdlet enables CredSSP authentication on a client or on a server computer. When CredSSP authentication is used, the user's credentials are passed to a

remote computer to be authenticated. This type of authentication is designed for commands that create a remote session from within another remote session. For example, you use this type of authentication if you want to run a background job on a remote computer.

Disable-WSManCredSSP

The Disable-WSManCredSPP cmdlet disables CredSSP authentication on a client or on a server computer.

There are other WSMan cmdlets introduced in PowerShell 2.0 such as Connect-WSMan, Disconnect-WSMan, Get-WSManInstance, New-WSManInstance, New-WSManSessionOption, Remove-WSManInstance and Set-WSManInstance. These cmdlets are not really meant for PowerShell remoting but we will discuss them as required.

Chapter 2: Enable/Disable PowerShell remoting

Remoting in PowerShell 2.0 can be enabled by just running the following cmdlet in an elevated PowerShell prompt

Enable-PSRemoting

Yes. That is it. You will be asked to respond to a couple of questions — based on OS architecture – as you see in the screenshot here.

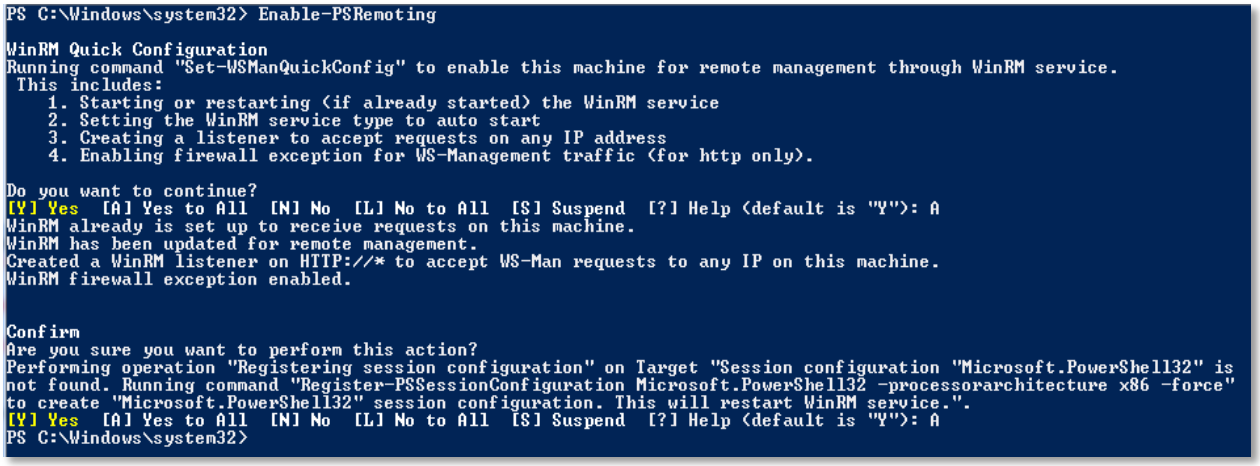


Figure 1 Enable Remoting

The following things happen when you run this cmdlet.

1. WinRM service gets enabled and startup type is set to auto start.
2. WinRM listener gets created to accept remoting requests on any IP addresses assigned to local computer
3. Windows firewall exceptions for WinRM service will be created. This is essentially the reason why network location cannot be set to public if you want to enable PS remoting. Windows firewall exceptions cannot be enabled if the network location is set to public.
4. Enables all registered PS session configurations. We will talk about this in detail later.

By default, WinRM only enables http transport for accepting remoting requests. You can manually enable https transport using either winrm command or New-WSManIntance cmdlet. For now, let us not overwhelm with so much information. We will look at this in part 2 of this guide.

Note

By default, PowerShell remoting uses port number 5985 (for http) and 5986 (for https). This can be changed by modifying wsman:\Localhost\listener\listener*\port to a different value using Set Item cmdlet. However, beware that this will change port number for every WinRM listener on the system.

You should always use the more comprehensive Enable-PSRemoting cmdlet. You can use -force parameter along with this cmdlet to silently enable remoting.

Trivia

PowerShell remoting cannot be enabled remotely ☺

Test PowerShell remoting

You can use the Enter-PSSession cmdlet to test if remoting is enabled on the local machine or not.

```
Enter-PSSession -ComputerName localhost
```

If remoting is enabled and functional, you will see the prompt changing to something like this

Figure 2 Enter-PSSession on localhost

Note

A PowerShell session (PS Session) is an environment to run the remote commands and scripts. PowerShell 2.0 provides various cmdlets to manage these sessions. To see a list of all PSSession cmdlets, use ***Get-Command –noun PSSession***.

There is also a ***New-PSSessionOption*** cmdlet to change default behavior of a PS session. New-PSSession and Enter-PSSession cmdlets have a parameter, -sessionOption, to specify custom session options. You can use this to specify options such as

IdleTimeOut
Determines how long the PSSession stays open if the remote computer does not receive any communication from the local computer, including the heartbeat signal. When the interval expires, the PSSession closes.

OpenTimeOut
Determines how long the client computer waits for the session connection to be established. When the interval expires, the command to establish the connection fails.

OperationTimeOut
Determines the maximum time that any operation in the PSSession can run. When the interval expires, the operation fails.

SkipCACheck
Specifies that when connecting over HTTPS, the client does not validate that the server certificate is signed by a trusted certification authority (CA).

SkipCNCheck
Specifies that the certificate common name (CN) of the server does not need to match the hostname of

the server. This option is used only in remote operations that use the HTTPS protocol.

SkipRevocationCheck
Does not validate the revocation status of the server certificate.

Remoting in workgroup environments

You will not be able to connect to a computer in workgroup just by running Enable-PSRemoting cmdlet. This is essentially because the security levels on a workgroup joined computer are more stringent than on a domain joined computer. So, on workgroup joined computers, you need to enable a few more things before you can create remoting sessions.

On Windows XP

You need to make sure the local security policy to enable classic mode authentication for network logons. This can be done by opening "Local Security Policy" from Control Panel -> Administrative Tools. Over there, navigate to **Local Policies -> Security Options** and double click on **"Network Access: Sharing and Security Model for local accounts"** and set it to classic.

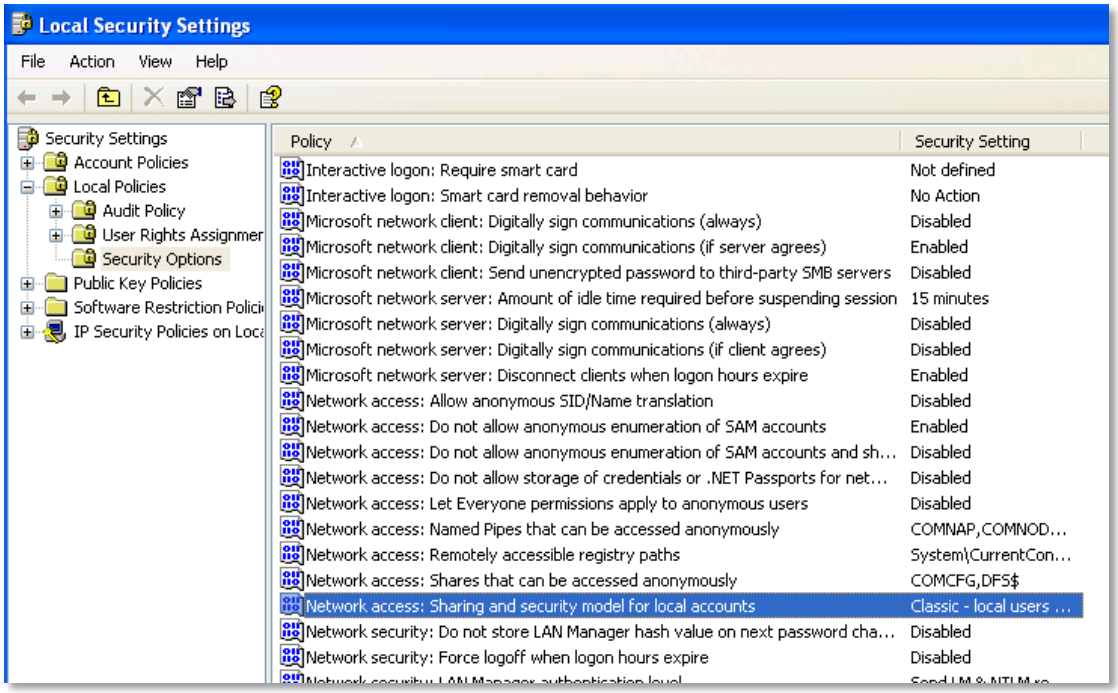


Figure 3 Windows XP security policy change

Modify WSMan trusted hosts setting

On all the workgroup joined computers – including Windows XP, Windows Vista and later – you need to add the IP addresses of all remoting clients to the list of trusted hosts. To do this,

```
Set-item wsman:localhost\client\trustedhosts -value *
```

Using * as the value will add all computers as trusted hosts. If you want to add only a specific set of computers,

```
Set-item wsman:localhost\client\trustedhosts -value Computer1,Computer2
```

If you want to add all computers in a specific domain,

```
Set-item wsman:localhost\client\trustedhosts -value *.testdomain.com
```

If you want to add an IP address of a remote computer to the trusted hosts list,

```
Set-item wsman:localhost\client\trustedhosts -value 10.10.10.1
```

Once the above changes are made, you can use Enable-PSRemoting cmdlet to enable remoting on these workgroup joined computers.

Remoting in mixed domain environment

By default, a user from a different domain cannot connect to a computer in another domain even when the user is a part of local administrators group. This is because remote connections from other domains run with only standard user privileges.

To work around this, you can change the LocalAccountTokenFilterPolicy registry entry (set it to 1) to allow members of other domain to remote in to the local computer.

```
new-itemproperty -name LocalAccountTokenFilterPolicy -path `
HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System -propertyType DWord -value 1
```

Disable remoting

You can use Disable-PSRemoting to disable remoting on the local computer. Disable-PSRemoting will only disable the session configurations. This will *not* remove all the changes done by Enable-PSRemoting. This includes leaving the WinRM service in enabled state and leaving all the listeners created to enable PS remoting. You will have to manually undo these changes if they are not required by any other component or service on the local computer.

If no other service or components on the local computer need WinRM service, you can disable WinRM service by running

```
Set-Service winrm -StartupType Manual
Stop-Service winrm
```

To remove all WinRM listeners listening on default PS remoting port (5985)

```
Get-ChildItem WSMan:\localhost\Listener –Recurse | Foreach-Object { $_.PSPath } | Where-Object { (Get-Item
"$_\Port").Value -eq 5985 } | Remove-Item
```

Enable remoting for only a specific network adapter

When you enable remoting on a computer using Enable-PSRemoting cmdlet, an http listener will be created to listen for remoting requests on all IP addresses on the local computer. This may not be a great security practice in an enterprise.

For example, you have an Internet facing server with two network connections. One – obviously – is the Internet connection and a second one connecting to your internal network. You don't need remoting be enabled on the network adapter connected Internet. But, since you used Enable-PSRemoting cmdlet, remoting will be enabled and there is a WinRM listener on the Internet facing network too. **So, how do we disable remoting on the Internet facing adapter?**

Enable-PSRemoting is a comprehensive cmdlet that does lot of things for you in one shot. This is also the recommended way to enable remoting. So, if we need to disable remoting on a particular IP address, all you need to do is remove the WinRM listener create by Enable-PSRemoting cmdlet and re-create your own listener for a specified IP address.

We use Remove-WSManInstance and New-WSManInstance cmdlets to do this. You can also use winrm command-line to achieve this. It is just a preference.

To remove the http listener created by Enable-Remoting,

```
Remove-WSManInstance winrm/config/Listener -SelectorSet @{Address="*";Transport="http"}
```

This will remove the listener.

Now, to re-create the http listener on a specified IP address

```
New-WSManInstance winrm/config/Listener -SelectorSet @{Address="IP:192.168.100.2";Transport="http"}
```

Once this listener is created successfully, you need to restart the WinRM service using Restart-Service cmdlet. From this point onwards, system will listen only on 192.168.100.2 IP address for any remoting requests.

You can follow the same approach for HTTPS transport too. However, you will have to specify the CertificateThumbPrint.

Remoting in an enterprise

To enable remoting on multiple computers in an enterprise or domain environment, you can use group policy[5]. For more information on this, refer to Appendix B: Enable PowerShell remoting using group policy

Summary

In this chapter, we looked at how to enable remoting for basic usage. We also looked at how to configure computers in a workgroup and mixed domain environment to participate in PowerShell remoting. Beware that disabling remoting won't undo changes done by Enable-PSRemoting. If remoting is not required on the local computer, you should manually undo all the changes. This is a good security practice. There are few more things you should be aware of while configuring a computer for remoting. We will look at each of these in detail in part 2.

[5] http://technet.microsoft.com/en-us/library/dd347642.aspx

Chapter 3: Execute remote commands

Within remoting, there are a couple of ways to run commands or scripts on a remote machine. This includes Invoke-Command cmdlet and interactive remoting sessions. These two methods deserve a detailed discussion for each and hence we will see Invoke-Command method in this chapter and discuss interactive remoting in the next chapter.

Once you have enabled remoting on all your computers, you can use Invoke-Command cmdlet to run commands and scripts on local computer or on remote computer(s). There are many possible variations of this cmdlet.

Run script blocks on local or remote computer

You can invoke a command on local or remote computer(s) using the below method

```
Invoke-Command -ComputerName SP2010-WFE -ScriptBlock {Get-Process}
```

The ScriptBlock parameter can be used to specify a list of commands you want to run on the remote computer. ComputerName parameter is not required for running commands on the local machine. If you want to run the same command on multiple remote computers, you can supply the computer names as a comma separated list to ComputerName parameter or use a text file as shown in the example here

```
Invoke-Command -ComputerName SP2010-WFE, SP2010-DB -ScriptBlock {Get-Process}
```

OR

```
Invoke-Command -ComputerName (get-content c:\scripts\servers.txt) -ScriptBlock {Get-Process}
```

This method is also called fan-out or 1: many remoting. You run the same command on multiple computers in just a single command.

All commands and variables in the ScriptBlock are evaluated on the remote computer. So, if you do something like -ScriptBlock {Get-Process -Name $procName}, PowerShell expects the remote computer session to have $procName defined. You can however pass variables on the local computer to a remote session when using Invoke-Command. This brings us to the next point in our discussion.

Run script files on remote computers

There are a couple of ways of doing this.

First method is to use the –FilePath parameter.

```
Invoke-Command -ComputerName SP2010-WFE -FilePath C:\scripts\Test.PS1
```

Note that the script you provide as a value to -FilePath must exist on the local machine or at a place accessible to the local machine. So, what if you want to run a script that exists only on the remote server?

You can use -scriptblock for that.

```
Invoke-Command -ComputerName SP2010-WFE -scriptBlock { C:\scripts\Test.PS1 }
```

This way, you can execute the script present on a remote machine but not on the local system.

Passing variables to remote session

Taking the above example, we can pass name of the process you are looking for as a variable to the script block. ArgumentList parameter helps you achieve this. You can do this as shown here.

```
$procName = "powershell"
Invoke-Command -ComputerName (get-content c:\scripts\servers.txt) `
-ScriptBlock {param ($Name) Get-Process -Name $Name} –ArgumentList $procName
```

The above example may be a simple one but it shows how to use -ArgumentList parameter to pass local variables to the remote session.

Using persistent sessions with Invoke-Command

Whenever you run Invoke-Command with -ComputerName parameter, a temporary session gets established to execute the remote command.

So, establishing a session every time you use this cmdlet can be time consuming. This may be OK for executing a couple of commands but not when you want to execute more commands or scripts. So, to avoid this we can use a persistent session to the remote computer and that is what -Session uses. You can create a persistent connection to a remote computer by using New-PSSession cmdlet as shown here.

```
$s = New-PSSession -ComputerName SP2010-WFE
```

Now, $s contains the session details for the persistent connection. We can use $s to invoke a command on the remote computer and the syntax for that will be

```
Invoke-Commad -Session $s -ScriptBlock {get-Process}
```

$s contains all the variables you create / modify when you execute commands on the remote computer. So, subsequent command execution with $s as the session will have access to all of the variables created / updated on the remote computer. For example,

```
$s = new-pssession -computername SP2010-WFE
Invoke-Command -Session $s -ScriptBlock {$fileCount = (Get-ChildItem D:\ -Recurse).Count}
invoke-command -session $s -scriptblock {$fileCount}
```

We could access $fileCount variable only because we used a persistent session to run the command. This would not have been possible if used -ComputerName to invoke the remote command.

Running remote command as a background job

The example shown above — which gets the total file count on D:\ of a remote machine — can be quite time consuming based on how big is D:\ on the remote computer. In such case, you will have to wait for

the remote command to complete execution. To avoid this, you can use -AsJob parameter to run the command as a background job[6] on the remote computer.

```
Invoke-Command -ComputerName SP2010-WFE -ScriptBlock {(Get-ChildItem D:\ -Recurse).Count} –asJob
```

Once you run this, you will see the job details listed as shown here

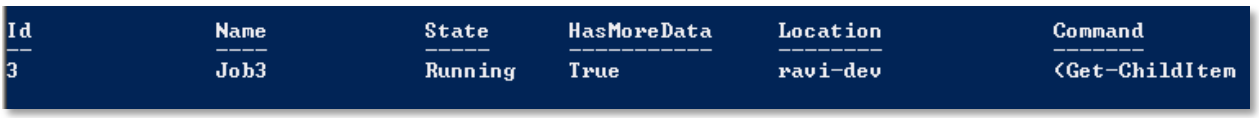


Figure 4 Background Job

When you use –asJob parameter with Invoke-Command cmdlet, the background job gets created locally but runs on the remote computer. Since this job is created locally, we can use *-Job cmdlets to manage the job object.

For example, you can use Get-Job to monitor the status of the job and once the job status changes to completed, you can use Receive-Job cmdlet to see the output of the script block specified.

```
Get-Job -Id 3 | Receive-Job
```

You can also use Start-Job within the script block to create a background job on the remote computer. However, this way the job output will be available only on the remote computer. So, when you need to get the output from this background job, you need to use Receive-Job within the script block to Invoke-Command.

For a complete discussion on background jobs in remoting, refer to About_Remote_Jobs article on technet.

Specifying credentials required for remoting

As discussed earlier in chapter 2, you can use PowerShell remoting between computers in a workgroup environment. All of the examples I showed above assume that you have access to remote computer as an administrator. This method works quite well in a domain environment where the logged on user has administrator credentials to access any computer in the domain. So, you don't have to explicitly pass the credentials to Invoke-Command. However, this will not work in a workgroup setup and you need to pass the credentials along with Invoke-Command. To do that,

```
$cred = Get-Credential
Invoke-Command -ComputerName SP2010-WFE -ScriptBlock { Get-Process} -Credential $cred
```

In the example above, Get-Credential prompts for the credentials to access remote computer and uses the same while calling Invoke-Command cmdlet.

Note

[6] About Background jobs: http://technet.microsoft.com/en-us/library/dd315273.aspx

Summary

In this chapter, we looked at how Invoke-Command cmdlet can be used to execute commands on a remote computer. We looked how to create a persistent session and use that along with Invoke-Command. You can use background jobs in remoting to execute time consuming commands as a job on the remote machine. There are other parameters to Invoke-Command include session options, etc. We will look at this in detail in part 2 of this guide.

Chapter 4: Interactive remoting sessions

To understand the advantages of interactive remoting in PowerShell 2.0, let us first look at some gotchas with Invoke-Command. Take an example of a remote system where SharePoint 2010 is installed. SharePoint 2010 provides native PowerShell cmdlets and these cmdlets can be accesses only if you load Microsoft.SharePoint.PowerShell PS snap-in. So, to do this using Invoke-Command

```
$s = New-PSSession -ComputerName SP2010-WFE

#load the PS Snap-in to enable SharePoint PS cmdlets
Invoke-Command -Session $s -ScriptBlock {Add-PSSnapin Microsoft.SharePoint.PowerShell}

#$s has the PowerShell cmdlets now
Invoke-Command -Session $s -ScriptBlock {Get-SPWeb http://sp2010-wfe:999}
```

If you look at the above code, we will have to use a persistent session so that we can use SharePoint cmdlets in subsequent Invoke-Command calls. Without a persistent session, you will have to load the SharePoint snap-in every time before using a SharePoint cmdlet.

Another caveat will be the unavailability of remote computer cmdlets in the local PowerShell session — in this case, the SharePoint 2010 cmdlets. This — essentially – means that we cannot use Get-Help or Get-Command cmdlets against the SharePoint 2010 cmdlets in the local session unless we pass that as a script block to Invoke-Command.

One more disadvantage of using Invoke-Command is unavailability of command completion. Unless the cmdlet you are using inside the script block is available locally, you cannot use tab completion. This can be a pain for many, including me.

This is where interactive remoting comes in to play.

Starting an interactive remoting session

Enter-PSSession and Exit-PSSession are the cmdlets used to start / exit an interactive remoting session. To enter an interactive session,

```
Enter-PSSession -ComputerName Ravi-Dev
```

Once you enter an interactive remoting session, PowerShell prompt changes to reflect the remote computer name you just connected to. This indicates that you are in an interactive remoting session.

Figure 5 Interactive Session

You can now add the SharePoint snap-in using Add-PSSnapin cmdlet.

```
Add-PSSnapin Microsoft.SharePoint.PowerShell
```

Once the snap-in is loaded, you will have access to all the SharePoint 2010 cmdlets as if they are available on the local computer. You can verify that by using Get-Help against one of the SharePoint 2010 cmdlets.

```
Get-Help Get-SPWeb –Full
```

Exiting an interactive session

You can use Exit-PSSession to come out of an interactive PS Session. Remember, by specifying –ComputerName as the parameter to Enter-PSSession, we are using only a temporary PS Session and is not a persistent session. So, any variables you create and the command history will all be gone if you exit this interactive session.

Using persistent sessions with interactive remoting

As discussed earlier, it will be advantageous to use persistent sessions. By using a persistent session, you can enter and exit the interactive session as many times as you like. All the data and variables you created in the remote session will persist until you remove the session. You can do it the same way you used persistent sessions with Invoke-Command.

```
$s = New-PSSession -ComputerName SP2010-WFE
Enter-PSSession -Session $s
```

Starting interactive remoting with an existing session

It is quite possible that you would have created a persistent session to use with Invoke-Command. You can use the same persistent session with Enter-PSSession to start an interactive remoting session.

You can use Get-PSSession cmdlet to see a list of all available/opened PS Sessions and then use Enter-PSSession as shown above to start interactive remoting. As you see here, I will pipe Get-PSSession output to Format-List cmdlet to get all session details.

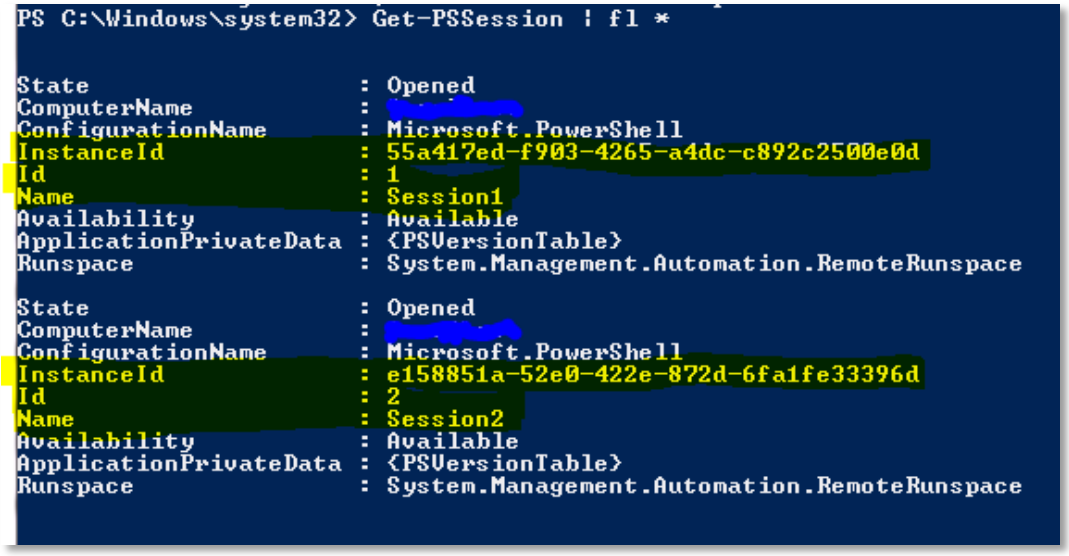


Figure 6 Get-PSSession

There are four ways to enter an existing PS Session for interactive remoting. I have highlighted the available options in the above screenshot. You can use whichever way is convenient to you.

Method 1: Using session Id

```
Enter-PSSession -id 1
```

Method 2: Using session instance Id

```
Enter-PSSession -InstanceId 55a417ed-f903-4265-a4dc-c892c2500e0d
```

Method 3: Using session name

```
Enter-PSSession -Name Session1
```

Method 3: Using –session parameter

```
$s = Get-PSSession -Id 1
Enter-PSSession -Session $s
```

All of the above options start interactive session using the persistent session “session1”. It is just more than one way to do the same thing.

Summary

This chapter gave a quick overview of interactive remoting in PowerShell 2.0 and how to use Enter-PSSession, Exit-PSSession and Get-PSSession cmdlets.

Chapter 5: Implicit remoting in PowerShell

In chapter 4 on interactive remoting sessions, we looked at how we can enter a remote session and then execute commands as if they were local. However, if you'd observed it more closely, we were actually sitting in the remote session than local session. The change in PowerShell prompt indicates this fact clearly.

In this chapter, we will look at implicit remoting feature in PowerShell. This feature makes it possible to run the commands / scripts on the remote computer while in the local session. Just read on if that statement sounds confusing.

Why implicit remoting?

We use interactive remoting to overcome a few disadvantages of using Invoke-Command. This method too has its own drawbacks. Within interactive remoting, you explicitly enter/exit a remote session. This also means that you are connected only to one remote computer and you have access only to the cmdlets or modules available on that remote computer. What if you want to access different cmdlets available on different computers?

For example, let us say you have two different computers one with Exchange 2010 and other with SharePoint 2010. Now, if you want to access cmdlets available to manage both these technologies from a "single computer" and in the "local session". Take a note, "single computer" and "local session" is the key to understand the concept of implicit remoting. The important thing to understand is that we need to manage multiple computers / technologies without ever the need to go out of local PowerShell session.

Using Invoke-Command is certainly not the choice because it involves setting up a session to the remote computer and then sending a script block to execute in that session. This is quite tedious. Although interactive remoting can eliminate the drawbacks of Invoke-Command, it is specific one remote session. So, if you are connected to the Exchange 2010 remote session, your SharePoint 2010 session is not available. This is where implicit remoting becomes important.

Implicit remoting can be used to bring remote commands to a local session. In implicit remoting, once you import remote commands in to a local session, you don't have to worry about the PS session details. You can import any number of remote sessions in to the local session making it possible to access cmdlets from different product technologies in the same local session. PowerShell will take care of that for you in the background.

Creating an implicit remoting session

Well, we have to first create a persistent PS session using New-PSSession and then use that to import remote commands in to local session. You can do it as shown here

```
$s = New-PSSession -ComputerName SP2010-WFE
Import-PSSession -Session $s
```

By default, Import-PSSession imports all commands except for commands that have the same names as commands in the current session. To import all the commands, use the -AllowClobber parameter. You will see a progress bar on top of the console window showing the progress of the import.

If you import a command with the same name as a command in the current session, the imported command hides or replaces the original commands. This is because import session converts the cmdlets to functions before importing and functions take precedence over cmdlets. So, imported commands take precedence over the local commands with same name -- irrespective of the fact whether those commands are loaded after importing a session or before.

To know more about the command precedence, read about_Command_Precedence.

However, aliases are an exception. Original aliases in the local session take precedence over imported aliases.

Avoiding name conflicts while importing a remote session

Import-PSSession provides a -Prefix parameter which adds the specified prefix to the nouns in the names of imported commands. For example,

```
Import-PSSession -Session $s -Prefix RS
```

This will prefix RS to all the cmdlets imported from a remote computer. So, if Get-Command was imported using this method, the local session will have Get-RSCommand and when you use this cmdlet, PowerShell implicitly runs this command inside the remote session.

As we discussed earlier in this chapter, PowerShell manages implicit remoting in the background. So, the behavior of Invoke-Command, creating/destroying a PS session every time we execute a remote command, exists with implicit remoting too. Hence, you will see that executing remote commands over this method a bit slow. To work around this, import-PSSession adds a -asJob parameter to all the commands imported in to the local session.

For example,

```
$s = New-PSSession -ComputerName Ravi-Dev
Import-PSSession -Session $s -Prefix RS
Get-RSProcess -asJob
```

This will run Get-RSProcess on the remote computer as a background job. Make a note that the original Get-Process has no -asJob parameter.

Importing modules and snap-ins to local session

```
$s = New-PSSession -ComputerName Ravi-Dev
Invoke-Command -Session $s -ScriptBlock {Import-Module ActiveDirectory}
Import-PSSession -Session $s -Module ActiveDirectory
```

In the above example we first create a PS session, import active directory module using Invoke-Command and then import the session in to the local session. This makes all the active directory cmdlets available in the local session.

Now, we can connect do a different remote session and import cmdlets from that session also.

```
$s = New-PSSession -ComputerName SP2010-WFE
Invoke-Command -Session $s -ScriptBlock {Add-PSSnapin Microsoft.SharePoint.PowerShell}
Import-PSSession -Session $s
```

Now, within the local session, we have access to AD cmdlets from one computer and SharePoint 2010 cmdlets from another machine. This makes it easy to manage both from the same computer and local session without worrying much about creating / destroying sessions.

Limitations of implicit remoting

Using implicit remoting you cannot import variables or Windows PowerShell providers. You cannot start a program with user interface or requires access to interactive desktop. Since, import-PSSession uses Invoke-Command to run the remote commands, it may be slow. Hence, all imported commands get support for –asJob parameter to run them as background jobs on the remote computer.

Summary

Implicit remoting is about bringing remote commands to local session. This technique can be used to import modules/snap-ins for commands that aren't available natively to PowerShell. In this chapter, we looked at how to create implicit remoting sessions and a few parameters used along with Import-PSSession cmdlet.

Chapter 6: Saving remote sessions to disk

In chapter 5, we looked at how we can use Import-PSSession cmdlet to execute remote commands as if they were local. This is nice but this will last only while the persistent session is alive. The moment we kill the session — using Remove-PSSession or the session is broken; the implicit remoting session will also get killed.

In this chapter, we look at how we can save a remoting session to disk so that we don't even have to explicitly create a PS session to execute commands on a remote computer.

Export remote session to a module on disk

This is achieved using Export-PSSession cmdlet. This cmdlet lets us import commands from a remote session and save the same in a PowerShell module on the local disk. This cmdlet can get cmdlets, functions, aliases, and other command types in to a PowerShell module. The following example shows how we can achieve this.

```
$s = New-PSSession -ComputerName SP2010-WFE
Invoke-Command -Session $s -ScriptBlock {Import-Module ActiveDirectory}
Export-PSSession -Session $s -OutputModule ADRemoteCommands -AllowClobber -Module ActiveDirectory
```

In the above example, the first two lines should be quite familiar by now. The third line is where the magic happens. We tell Export-PSSession cmdlet to export all the commands, aliases, functions, etc available in PS Session $s to a module on hard disk and name it ADRemoteCommands.

If the Export-PSSession is successful, you will see output similar to what is shown here

```
    Directory: C:\Users\administrator\Documents\WindowsPowerShell\Modules\ADRemoteCommands

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          2/4/2010   9:43 PM     196132 ADRemoteCommands.psm1
-a---          2/4/2010   9:43 PM         99 ADRemoteCommands.format.ps1xml
-a---          2/4/2010   9:43 PM        597 ADRemoteCommands.psd1

PS C:\Users\administrator>
```

Figure 7 Export-PSSession output

In the above output, it is clear that Export-PSSession generates .psm1, .psd1 and format data file for the module automatically. Now, you can load the module at any later point in time to get access to the remote commands.

Importing a module saved on disk

If you observe the output closely, path where the module files are stored is same as $Env:PSModulePath. So, you don't need to specify the absolute path to the module.

```
Import-Module ADRemoteCommands
```

This imports all remote commands available in the module to local session. Whenever we execute a remote command, implicit remoting kicks in, establishes the remote session, executes the command in remote session and returns the output. All this is done without you really using any remoting related cmdlets. If establishing a remote session requires a password, you will be prompted for one.

Limitations of Export-PSSession

Using Export-PSSession has the same limitations as implicit remoting. You cannot use Export-PSSession to export a Windows PowerShell provider. You cannot start a program with user interface or requires access to interactive desktop. The exported module does not include the session options used to create the session. So, if you need any specific session options to be configured before running remote commands, you need to create a PS Session with all the required session options before importing the on disk module.

Summary

Saving a remote session to disk can be done using Export-PSSession cmdlet. This is a very quick method to execute commands on a remote computer without explicitly creating a PS session or entering an interactive remoting session.

This also brings us to the end of part 1. In part 1, we looked at the basics of PowerShell remoting such as what is remoting, enabling remoting in various scenarios, executing remote commands and importing/exporting remoting sessions. Part 2 of this guide looks at a bit more advanced aspects of PowerShell remoting.

Keep reading..!

PART 2

Chapter 7: Understanding session configurations

In chapter 2, we saw that whenever PowerShell remoting is enabled, the default session configurations get registered. Also, Invoke-Command, Enter-PSSession and New-PSSession cmdlets have a –ConfigurationName parameter which can be used to specify a different session configuration than the default one. So, what are these session configurations?

So, in this part, we will look at all the PS session configuration cmdlets; discuss how to create custom PS Session configurations and the need for it. Let us dive in to this now.

What is a PS Session configuration?

A session configuration can be used to define

- Who can create a Windows PowerShell session on the local computer
- What level of access — to cmdlets, scripts and PowerShell language — they have on the local computer, etc.

When you enable PowerShell remoting using Enable-PSRemoting, you will see a final step performing Microsoft.PowerShell and Microsoft.PowerShell32 (on x64 systems) session configuration registrations. These default session configurations are used when the remote users connecting to local system do not specify a configuration name. By default, only members of administrators group have access to these two session configurations. Hence, only members of administrators group will be able to create remoting sessions by default.

Based on the above description, PowerShell session configurations can be used to

- customize the remoting experience for users
- delegate administration by creating session configuration with varying levels of access to system

In this chapter, we will look at session configurations and see how we can create custom session configurations. We will discuss delegated administration at depth in a later chapter.

Cmdlets available to manage session configurations

The following cmdlets are available to manage session configuration.

1. Register-PSSessionConfiguration
2. Unregister-PSSessionConfiguration
3. Enable-PSSessionConfiguration
4. Disable-PSSessionConfiguration
5. Set-PSSessionConfiguration
6. Get-PSSessionConfiguration

Creating a new session configuration

Register-PSSessionConfiguration cmdlet can be used to create a new session configuration. You can use a C# assembly or a PowerShell script as a startup script for this new session configuration. This startup script can be used to customize the remoting experience. For example, create a script the imports Active Directory module using import-module cmdlet as shown here.

```
Import-Module ActiveDirectory
```

Save this script as startupscript.ps1 or any name of your choice on the local computer. Now, use the Register-PSSessionConfiguration cmdlet to create a new session configuration. This can be done by running

```
Register-PSSessionConfiguration -Name "ActiveDirectory" -StartupScript C:\scripts\StartupScript.ps1
```

You will be prompted to confirm this action and at the end to restart WinRM service on the local computer.

Note

You must enable script execution on the local computer to be able to use the startup script as a part of session configuration

List available session configurations

From the local computer

Get-PSSessionConfiguration cmdlet lists all the available session configurations on the local computer.

```
PS C:\scripts> Get-PSSessionConfiguration

Name                    PSVersion  StartupScript        Permission
----                    ---------  -------------        ----------
ActiveDirectory         2.0        C:\scripts\startu...
microsoft.powershell    2.0                             BUILTIN\Administrators AccessAll...
microsoft.ServerManager 2.0                             BUILTIN\Administrators AccessAll...
```

Figure 8 Get-PSSessionConfiguration

As you see in the above output, Get-PSSessionConfiguration lists all available session configurations on the local computer and who has permission to access the configuration. No permissions have been assigned yet to the new active directory configuration.

From a remote computer

Get-PSSessionConfiguration cmdlet cannot be used to access a list of PS Session configurations from a remote computer. However, we can use Get-WSManInstance cmdlet to achieve this.

```
Get-WSManInstance winrm/config/plugin -Enumerate -ComputerName SP2010-WFE | Where `
{ $_.FileName -like '*pwrshplugin.dll'} | Select Name
```

This will list all the session configuration names as available on the remote computer. You can then use one of the session configurations to connect to the remote computer using PowerShell remoting.

Note

You must be an administrator and run the above command at an elevated prompt. Also,
You must have access to the session configuration on the remote computer to be able to use it within PowerShell remoting.

Custom permissions and PS Session configurations

You can use Set-PSSessionConfiguration to allow access to invoke the new session configuration. To do this,

```
Set-PSSessionConfiguration -Name ActiveDirectory -ShowSecurityDescriptorUI
```

This opens up the dialog to add permissions to invoke this session configuration. As you see in the screenshot here, administrators group has no invoke permission on this session configuration.

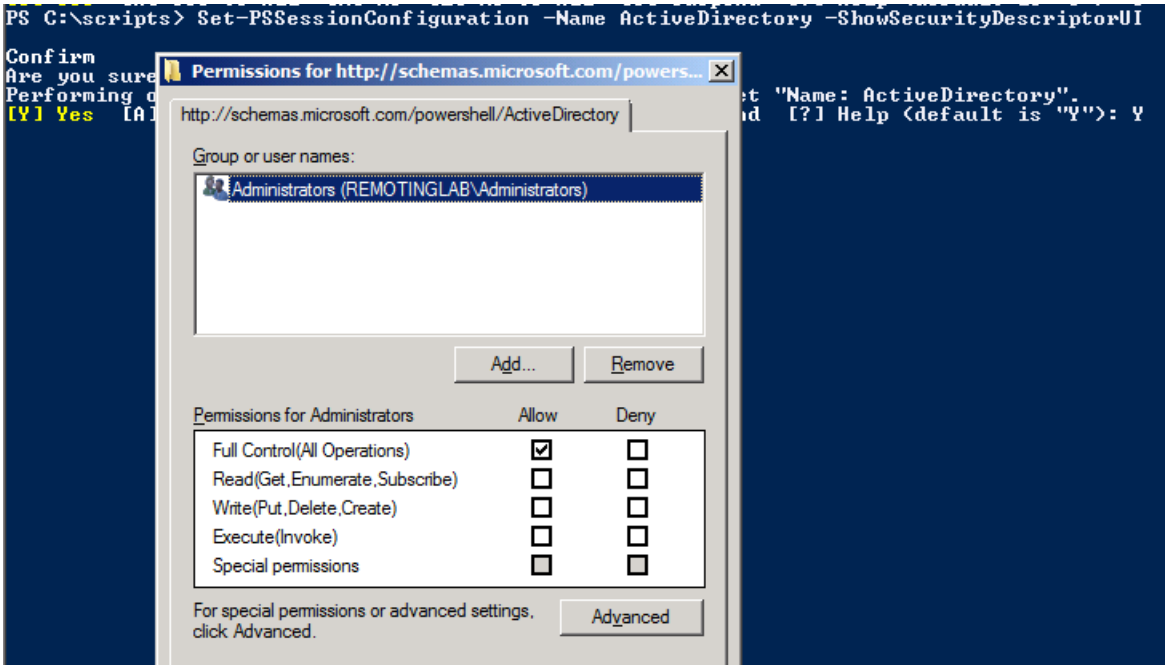


Figure 9 Security descriptor UI

Select **Allow -> (Execute) Invoke permission** and click OK. You will be prompted to restart the WinRM service. Now, an administrator or a member of administrators group will be able to use this session configuration. Similarly, you can add a non-administrator user to the list of users/groups and then assign appropriate permissions. This way, you can have non-administrator uses to remote in to the local computer using PowerShell remoting. You can read more on this in the next chapter.

Invoking a custom session configuration

You can use New-PSSession, Enter-PSSession and Invoke-Command cmdlets to load a session configuration other than the default configuration. The ConfigurationName parameter can be used to specify the session configuration. The following code snippet shows three different ways to invoke a remote session using a custom session configuration name.

```
$s = New-PSSession -ComputerName SP2010-WFE -ConfigurationName ActiveDirectory

Enter-PSSession -ComputerName SP2010-WFE -ConfigurationName ActiveDirectory

Invoke-Command -ComputerName SP2010-WFE -ConfigurationName ActiveDirectory -ScriptBlock {Get-Process}
```

Note

To be able to use a StartupScript, script execution policy must be set to an appropriate setting on the local computer where the session configuration is registered.

In an earlier chapter, we used Invoke-Command to load the active directory module within a persistent session and then use that persistent session to import active directory cmdlets in to local session. However, by using a session configuration that import active directory module as a startup script, we will have all the AD cmdlets available as soon as we connect to the remote session.

Disable a session configuration

You can use Disable-PSSessionConfiguration cmdlet to disable an existing session configuration and prevents users from connecting to the local computer by using this session configuration. You can use -Name parameter to specify what session configuration you want to disable. If you do not specify a configuration name, the default Microsoft.PowerShell session configuration will be disabled.

The Disable-PSSessionConfiguration cmdlet adds a "deny all" setting to the security descriptor of one or more registered session configurations. As a result, you can unregister, view, and change the configurations, but you cannot use them in a session. Disable-PSRemoting cmdlet will disable all PS Session configurations available on the local computer.

Enable-PSSessionConfiguration cmdlet can be used to enable a disabled configuration. You can use -Name parameter to specify what session configuration you need to enable.

Delete a session configuration

You can use Unregister-PSSessionConfiguration cmdlet to delete a previously defined session configuration. It is quite possible to delete the default session configuration — Microsoft.PowerShell — using this cmdlet. However, this default session configuration gets re-created if you re-run Enable-PSRemoting cmdlet.

Summary

In this chapter, we looked at the basics of PowerShell session configurations and how to create custom configurations. We also looked at cmdlets to manage these session configurations. By default, it is necessary that you need to a part of local administrators group to remote in to computer. However, using custom session configuration and permissions assigned to these configurations, we can enable a non-administrator user to remote in to a computer using PowerShell remoting.

Chapter 8: Using custom session configurations

“With great power comes great responsibility”, said Uncle Ben.

But some people don’t just understand that. That is when you have to rip-off their powers. Similarly, the default PS Session configuration allows full access to PowerShell language, cmdlets, scripts and everything available to PowerShell. Of course, you need to authenticate as a local administrator or should have execute permission to invoke the session. Running a few cmdlets such as Stop-Service or Restart-Computer can be quite dangerous on a production server. This is where a custom session configuration can help provide role based access to remote host using PowerShell remoting.

We touched upon creating custom session configuration in the previous chapter of this PowerShell remoting guide. In this chapter, we look at how we can extend the concept of custom session configuration to restrict available commands and PowerShell language in a remote session. I will go straight in to the startup script used to implement this since we already looked at how to create custom session configuration and assign permissions to a specific user.

```
$RequiredCommands = @("Get-Command",
                      "Get-FormatData",
                      "Out-Default",
                      "Select-Object",
                      "Out-file",
                      "Measure-Object",
                      "Exit-PSSession"
                      )

 $ExecutionContext.SessionState.Applications.Clear()
$ExecutionContext.SessionState.Scripts.Clear()

Get-Command -CommandType Cmdlet, alias, function | ?{$RequiredCommands -notcontains $_.Name}
| %{$_.Visibility="Private"}
$ExecutionContext.SessionState.LanguageMode="RestrictedLanguage"
```

As you see here, we have only a few required commands. We don’t want the remote user to execute commands other than this set. BTW, this set is the absolute minimum required even to start remoting session. So, consider this as a standard required commands list. Towards the end, we set the language mode to restrict to make sure the remote user cannot execute infinite loops, etc that could potentially bring the system down. This script, when used as the startup script for a session, will result in something as shown here.

```
PS C:\Windows\system32> Enter-PSSession -ComputerName Localhost -ConfigurationName Restricted
[localhost]: PS>Get-Process
The term 'Get-Process' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the
 spelling of the name, or if a path was included, verify that the path is correct and try again.
    + CategoryInfo          :
    + FullyQualifiedErrorId : CommandNotFoundException

[localhost]: PS>Get-Command

CommandType     Name                                               Definition
-----------     ----                                               ----------
Cmdlet          Exit-PSSession                                     Exit-PSSession [-Verbose] [-Debug] [-ErrorAction...
Cmdlet          Get-Command                                        Get-Command [[-ArgumentList] <Object[]>] [-Verb ...
Cmdlet          Get-FormatData                                     Get-FormatData [[-TypeName] <String[]>] [-Verbos...
Cmdlet          Measure-Object                                     Measure-Object [[-Property] <String[]>] [-InputO...
Cmdlet          Out-Default                                        Out-Default [-InputObject <PSObject>] [-Verbose]...
Cmdlet          Out-File                                           Out-File [-FilePath] <String> [[-Encoding] <Stri...
Cmdlet          Select-Object                                      Select-Object [[-Property] <Object[]>] [-InputOb...
```

Figure 10 inside a restricted Session

As you see above, get-Command lists only the commands we have in the Required Commands list. However, if you have a large list of required commands, the method you have seen in the above code is not scalable. Instead, you can use a denied list of commands that is relatively small. For example, if you don't want your users to execute Stop-Process or Restart-Computer, your code will look like

```
$DeniedCommands = @("Stop-Process",
                    "Restart-Computer"
                    )

$ExecutionContext.SessionState.Applications.Clear()
$ExecutionContext.SessionState.Scripts.Clear()

Get-Command -CommandType Cmdlet, alias, function | ?{$DeniedCommands -contains $_.Name}
| %{$_.Visibility="Private"}
$ExecutionContext.SessionState.LanguageMode="RestrictedLanguage"
```

So, if you use this code for your startup script, you will see something like this

```
PS C:\Documents and Settings\Ravi> Enter-PSSession -ComputerName WIN-8JVNU641TMQ -ConfigurationName Restrcited
[win-8jvnu641tmq]: PS C:\Users\ravi\Documents> Get-Process | Select -first 3

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     35       5     1788       3872    43               448 conhost
    436      11     1756       3548    43               344 csrss
    168      10     1640       4456    41               384 csrss


[win-8jvnu641tmq]: PS C:\Users\ravi\Documents> Stop-Process
The term 'Stop-Process' is not recognized as the name of a cmdlet, function, script file, or operable program. Check th
e spelling of the name, or if a path was included, verify that the path is correct and try again.
    + CategoryInfo          :
    + FullyQualifiedErrorId : CommandNotFoundException

[win-8jvnu641tmq]: PS C:\Users\ravi\Documents> Restart-Computer
The term 'Restart-Computer' is not recognized as the name of a cmdlet, function, script file, or operable program. Chec
k the spelling of the name, or if a path was included, verify that the path is correct and try again.
    + CategoryInfo          :
    + FullyQualifiedErrorId : CommandNotFoundException

[win-8jvnu641tmq]: PS C:\Users\ravi\Documents> _
```

Figure 11 Inside a restricted session

I prefer the second method.

If you need to extend or modify the behavior of commands in a remote session, you need to create command proxies. You can read more about it @

http://blogs.msdn.com/powershell/archive/2009/01/04/extending-and-or-modifing-commands-with-proxies.aspx

What I have shown here is just one way of achieving control in the remote sessions. However, based on your organization needs there could be a better way of doing this. These methods include user role based restrictions, etc as discussed at a PDC'09 session. Do refer to that for more information.

Chapter 9: Interpreting, formatting and displaying remote output

In this chapter, we will look at remoting output. This includes how the output is transferred from remote computer to local, how it is displayed and how we can format this output based on a need. We already discussed various methods to execute commands (part4, part 5 and part 6) on a remote computer. In this post, for the sake of our discussion of remoting output, I will use only Invoke-Command method to execute remote commands. However, I will point out the differences as required.

Note

Most of this does not apply within an interactive remoting session

The concepts of remoting output are explained in a TechNet article at http://technet.microsoft.com/en-us/library/dd347582.aspx . I am going to put some story around this to help you understand the concepts well.

First, let us start with an obvious difference in the output received from a remote session. If you use Invoke-Command to run Get-PSDrive, you see something like this.

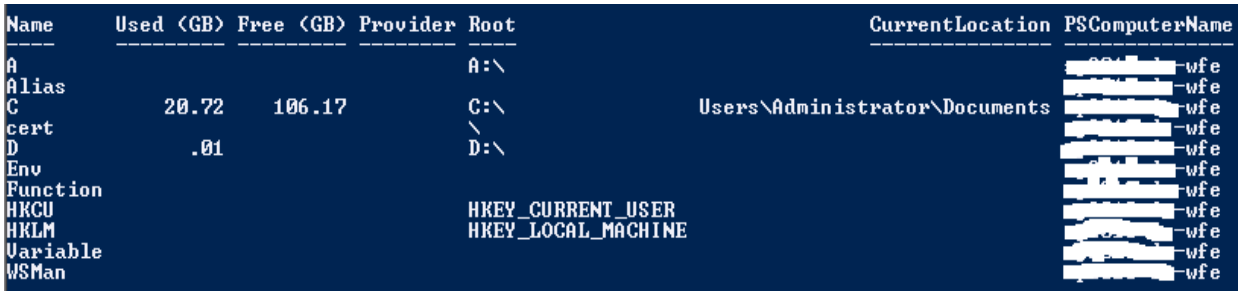


Figure 12 Remote Output

You can see an additional column in the output that shows the remote computer name with PSComputerName as the column name. This won't be displayed if you run the same cmdlet on local computer. So, if you don't want to display this information in the remote output you can use the -HideComputerName parameter.

It is also possible that some cmdlets may not display PSComputerName property. For example, Get-Date. In such a scenario you can add PSComputerName to the output of Get-Date as shown here

```
Invoke-Command -ComputerName Server01,Server02 -ScriptBlock {Get-Date} | ft DateTime, PSComputerName –Auto
```

```
DateTime                                PSComputerName
--------                                --------------
Monday, February 15, 2010 4:32:27 PM    [redacted]app
Monday, February 15, 2010 4:32:27 PM    [redacted]-wfe
```

Figure 13 Get-Date output

How remote output comes over to local computer?

The objects that Windows PowerShell cmdlets return cannot be transmitted over the network. So, the live objects are "serialized". In other words, the live objects are converted into XML representations of the object and its properties. Then, the XML-based serialized object is transmitted across the network to the local session where it gets de-serialized in to .NET object. This is how an MSDN article defines serialization in .NET framework.

> Why would you want to use serialization? The two most important reasons are, to persist the state of an object to a storage medium so an exact copy can be recreated at a later stage, and to send the object by value from one application domain to another. For example, serialization is used to save session state in ASP.NET and to copy objects to the clipboard in Windows Forms. It is also used by remoting to pass objects by value from one application domain to another.

As it is defined above, the live objects are converted in to XML based representation. So, once de-serialized in the local session, they don't expose any methods that actually belong to the object. Let us see an example to understand this. First, let us look at Get-Process output in a local session and see what all methods we see.

```
PS C:\Windows\system32> Get-Process | Get-Member -MemberType Method

   TypeName: System.Diagnostics.Process

Name                       MemberType Definition
----                       ---------- ----------
BeginErrorReadLine         Method     System.Void BeginErrorReadLine()
BeginOutputReadLine        Method     System.Void BeginOutputReadLine()
CancelErrorRead            Method     System.Void CancelErrorRead()
CancelOutputRead           Method     System.Void CancelOutputRead()
Close                      Method     System.Void Close()
CloseMainWindow            Method     bool CloseMainWindow()
CreateObjRef               Method     System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose                    Method     System.Void Dispose()
Equals                     Method     bool Equals(System.Object obj)
GetHashCode                Method     int GetHashCode()
GetLifetimeService         Method     System.Object GetLifetimeService()
GetType                    Method     type GetType()
InitializeLifetimeService  Method     System.Object InitializeLifetimeService()
Kill                       Method     System.Void Kill()
Refresh                    Method     System.Void Refresh()
Start                      Method     bool Start()
ToString                   Method     string ToString()
WaitForExit                Method     bool WaitForExit(int milliseconds), System.Void WaitForExit()
WaitForInputIdle           Method     bool WaitForInputIdle(int milliseconds), bool WaitForInputIdle()

PS C:\Windows\system32>
```

Figure 14 Local output

Here, you can see a list of methods you can use against a process object. Now, let us take a look at how this looks when we execute the same command in a remote session.

```
PS C:\Windows\system32> Invoke-Command -ComputerName Localhost -ScriptBlock {Get-Process} | Get-Member -MemberType Metho
d


   TypeName: Deserialized.System.Diagnostics.Process

Name     MemberType Definition
----     ---------- ----------
ToString Method     string ToString(), string ToString(string format, System.IFormatProvider formatProvider)
```

Figure 15 De-serialized output

If you observe in the above screenshot, TypeName represents a deserialized object and there are no methods that you can use against a process object. A deserialized object represents a snapshot of get-process at the time of command execution in the remote session. This also means that you can't execute methods such as Kill() against a deserialized process object. Also, no methods to modify the property set will work in the local session.

Windows PowerShell blog has a nice post on how objects are to and from a remote session. I recommend that you read this post for more information.

Formatting remote output

Most de-serialized objects are automatically formatted for display by entries in the Types.ps1xml or Format.ps1xml files. However, the local computer might not have formatting files for all of the de-serialized objects that were generated on a remote computer. When objects are not formatted, all of the properties of each object appear in the console in a streaming list. To get formatting data from another computer, use the Get-FormatData and Export-FormatData cmdlets. Again, let us take an example to understand this.

Take an example of a SharePoint 2010 farm and you want to access /run SharePoint 2010 cmdlets from a Windows 7 machine using Invoke-Command. First, if we run Get-SPSite on SharePoint 2010 web frontend, you will see

```
PS C:\Users\Administrator> Get-SPSite

Url
---
http://[redacted]
http://[redacted]-wfe:12121
http://[redacted]-wfe
```

Now, if we try to run the same in a remote session using Invoke-Command, you will see

```
PS C:\Users\administrator.SP2010LAB> Invoke-Command -Session $s -ScriptBlock {Get-SPSite}

PSComputerName                  : sp[redacted]wfe
RunspaceId                      : 1e27a299-cf00-4bc0-9c1b-5f7340549cd0
PSShowComputerName              : True
ApplicationRightsMask           : FullMask
ID                              : 753eaa68-98fc-4c77-b3b2-467339cbb076
SystemAccount                   : SHAREPOINT\system
Owner                           : SP2010LAB\administrator
SecondaryContact                :
GlobalPermMask                  : FullMask
IISAllowsAnonymous              : True
Protocol                        : http:
HostHeaderIsSiteName            : False
```

Figure 16 Remote Output

As you see in the above screenshot, the output from a remote session is quite different from the one you saw in a local session. This is because we don't have the formatting data available on the Windows 7 computer.

We can use Get-FormatData, Export-FormatData and Update-FormatData cmdlets to get the formatting data from a remote computer to local session. To do this

```
$s = New-PSSession -ComputerName SP2010-WFE
Invoke-Command -session $s -ScriptBlock {Add-PSSnapin Microsoft.SharePoint.PowerShell}
Invoke-Command -Session $s -ScriptBlock {Get-FormatData -TypeName *SharePoint*} | Export-FormatData -
Path C:\scripts\SharePoint.Format.ps1xml
Update-FormatData -PrependPath C:\scripts\SharePoint.Format.ps1xml
```

The above code snippet will let you import the formatting data for all SharePoint cmdlets in to the local session. Now, if we run Get-SPSite in the remote session using Invoke-Command,

```
PS C:\Users\administrator.SP2010LAB> Invoke-Command -Session $s -ScriptBlock {Get-SPSite}

Url                                                PSComputerName
---                                                --------------
http://m[redacted]                                 [redacted]-wfe
http://[redacted]-wfe:12121                        [redacted]-wfe
http://[redacted]-wfe                              [redacted]-wfe

PS C:\Users\administrator.SP2010LAB> Invoke-Command -Session $s -ScriptBlock {Get-SPSite} -HideComputerName

Url
---
http://[redacted]
http://[redacted]-wfe:12121
http://[redacted]-wfe
```

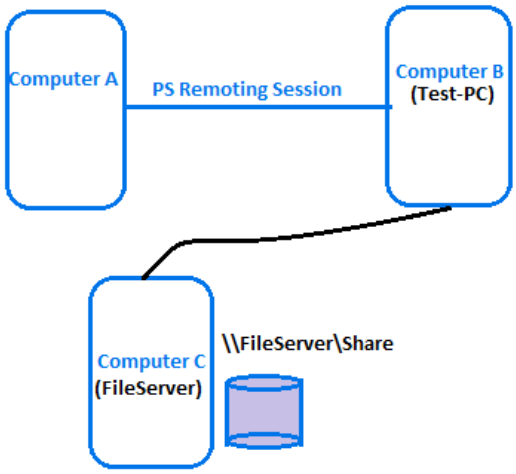
Figure 17 Remote output with formatting

Now, with the formatting information in the local session, you can see that Get-SPSite output is formatted similar to the one we saw when we ran the cmdlet in a local session. However, make a note that this applies only to the current session. If you close and re-open the PowerShell console, the formatting data will be lost. You can add the Update-FormatData cmdlet to your PowerShell profile to make the format data across all PowerShell sessions.

Chapter 10: Using CredSSP for multi-hop authentication

In this chapter, we will look at how CredSSP[7] can be used for multi-hop authentication[8] in PowerShell remoting. CredSSP and multi-hop support are not features of PowerShell 2.0 or PowerShell remoting, per se. Credential Security Service Provider (CredSSP) is a new security service provider that enables an application to delegate the user's credentials from the client to the target server. Multi-hop support in Windows Remote Management uses CredSSP for authentication. Since PowerShell 2.0 remoting is built on top of WinRM, we can use CredSSP to perform multi-hop authentication.

Let us look at an example to understand what multi-hop authentication is. Imagine a group of computers as shown here and you establish a remoting session from computer A (client) to computer B (server) and then from computer B, you try to create a file in a file share on computer C.



Now, within the remoting session to computer B, we want to execute a command — as below — to create test.txt on computer C.

```
Invoke-Command -ComputerName Test-PC.SP2010lab.com -credential SP2010LAB\Administrator -ScriptBlock {[System.IO.File]::Create(\\FileServer\Share\Test.txt)}
```

```
PS C:\Users\Administrator> Invoke-Command -ComputerName test-pc.sp2010lab.com -Credential SP2010LAB\Administrator -Scrip
tBlock {[System.IO.File]::Create("\\FileServer\Share\Test.txt")}
Exception calling "Create" with "1" argument(s): "Access to the path '\\FileServer\Share\Test.txt' is denied."
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : DotNetMethodException

PS C:\Users\Administrator> _
```

Figure 18 file share access error

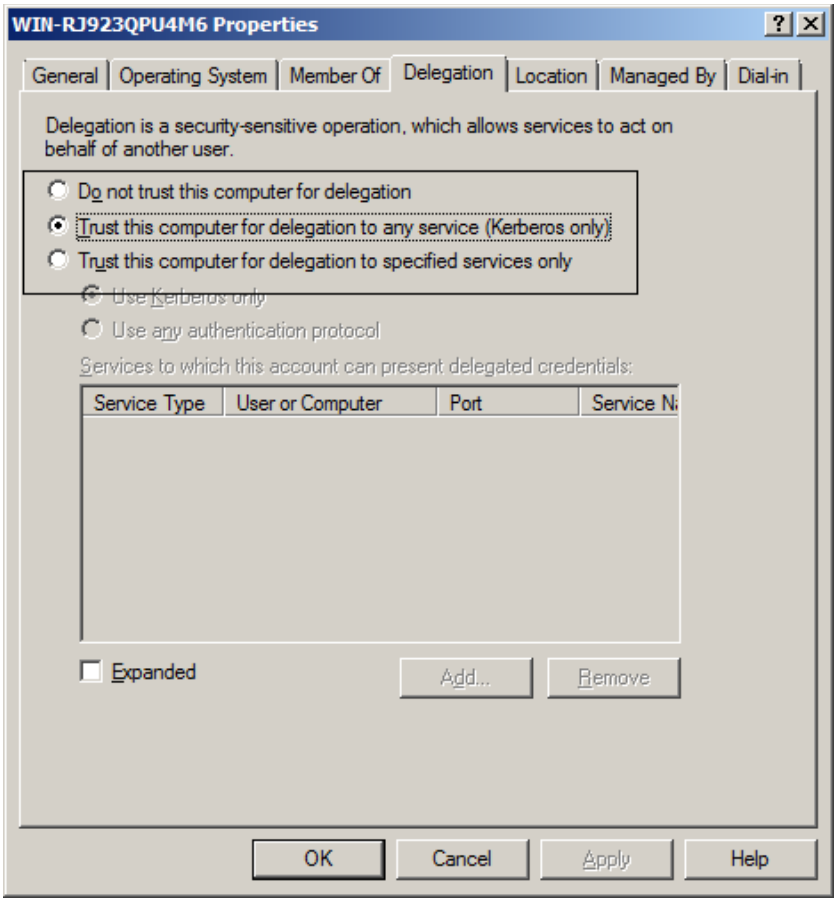
This command results in an "Access Denied" error as shown above. This command fails since the remote session tries to access the file share using the machine credentials instead of the credentials used to invoke the remote session. We could have successfully created the text file if there was a way to pass or

[7] CredSSP

[8] multi-hop authentication

delegate credentials from the client so that we can authenticate to the file share. This is what is called multi-hop authentication and PowerShell remoting enables this using CredSSP.

Note: When a domain controller (Windows Server 2008 or Windows Server 2008 R2 is used as a second hop, the credentials are always received and delegated without the need for CredSSP. Refer to https://connect.microsoft.com/PowerShell/feedback/details/630672/a-domain-controller-receives-and-delegates-credentials-even-when-credssp-is-not-configured for more information on this. The reason for this is the "Trust this computer delegation to any service (Kerberos only)" setting "Delegation Tab" of a Domain controller's properties in AD users and computer snap-in. This is enabled by default on all domain controllers.



Delegating credentials

The cmdlets to create a remoting session — Invoke-Command, Enter-PSSession and New-PSSession – have a parameter to specify the authentication method as CredSSP. However, before we use this parameter, we need to enable credSSP on the computers participating in multi-hop authentication. Also, when enabling CredSSP, we need to specify the role — client or server — of a computer. A client is the computer from which the remoting session is initiated and server is the computer from which the multi-hop authentication is triggered. So, from the above example, we need to enable CredSSP authentication on computer A and computer B.

PowerShell 2.0 provides the following cmdlets to manage CredSSP authentication.

1. Enable-WSManCredSSP
2. Disable-WSManCredSSP
3. Get-WSManCredSSP

Let us now look at how we enable WSManCredSSP and specify client / server roles. First, let us enable CredSSP on computer A.

Note

You need to run these cmdlets in an elevated prompt. And, these cmdlets are not available on Windows XP and Windows 2003. This means you cannot use the –CredSSP parameter also on a Windows XP or Windows 2003 computer.

```
Enable-WSManCredSSP -Role Client -DelegateComputer "*.SP2010lab.com"
```

As shown here, you can use Enable-WSManCredSSP cmdlet to enable CredSSP authentication and specify the computer role as client. When the computer role is defined as a client, you can also specify the DelegateComputer parameter to specify the server or servers that receive the delegated credentials from the client. The delegateComputer accepts wildcards as shown above. You can also specify "*" to specify all computers in the network.

When Enable-WSManCredSSP cmdlet is used to enable CredSSP on the client by specifying client in the role parameter. The cmdlet then performs the following:

1. The WS-Management setting <localhost|computername>\Client\Auth\CredSSP is set to true.
2. Sets the Windows CredSSP policy AllowFreshCredentials to WSMan/Delegate on the client.

Now, we will enable CredSSP on computer B and deginate that as server.

```
Enable-WSManCredSSP -Role Server
```

The above cmdlet enables CredSSP on computer B and sets the WS-Management setting <localhost|computername>\Service\Auth\CredSSP is to true. Now, we can use Invoke-Command to run the script block as shown at the beginning of this post. However, we will specify the authentication method as CredSSP and pass the credentials.

```
Invoke-Command -ComputerName test-pc.SP2010lab.com -Credential SP2010Lab\Administrator -Authentication CredSSP -ScriptBlock {[System.IO.File]::Create(\\FileServer\Share\Test.txt)}
```

```
PS C:\Users\Administrator> Invoke-Command -ComputerName test-pc.sp2010lab.com -Authentication CredSSP -Credential SP2010
LAB\Administrator -ScriptBlock {[System.IO.File]::Create("\\FileServer\Share\Test.txt")}


PSComputerName     : test-pc.sp2010lab.com
RunspaceId         : e118b24c-486c-4bf7-8744-ee4aaae28c06
PSShowComputerName : True
CanRead            : True
CanWrite           : True
CanSeek            : True
IsAsync            : False
Length             : 0
Name               : \\FileServer\Share\Test.txt
Position           : 0
Handle             : 808
SafeFileHandle     : Microsoft.Win32.SafeHandles.SafeFileHandle
CanTimeout         : False


PS C:\Users\Administrator>
```

Figure 19 CredSSP authentication

As you see here, we see the output from Create() method which shows the details of the newly created file.

Caution

CredSSP authentication delegates the user's credentials from the local computer to a remote computer. This practice increases the security risk of the remote operation. If the remote computer is compromised when credentials are passed to it the credentials can be used to control the network session.

You can use Disable-WSManCredSSP to disable CredSSP authentication on a client or a server computer.

```
Disable-WSManCredSSP -Role Client
```

```
Disable-WSManCredSSP -Role Server
```

You can use Get-WSManCredSSP cmdlet to verify if a computer has CredSSP enabled and also the role (client/server).

Summary

Credential delegation is required when you need to access or authenticate to a second computer within a remote session. Some of the other examples also include SharePoint 2010 cmdlets. Since, every SharePoint cmdlet will have to gather data from various servers within the farm and depending on how you farm accounts are configured, you may need to use CredSSP authentication to authenticate to all the servers within the farm.

This chapter concludes this remoting guide.

Appendix A: Frequently asked questions

1. **Is it mandatory that the user invoking a remote session has administrator privileges?**
   A: Not necessary. Any non-administrator user can start a remote session if he/she has invoke permissions to a session configuration on a remote computer. Refer to Chapter 7 for information on how to do this.
2. **Do I need PS remoting enabled on all the computers participating in remoting?**
   A: No. You need to enable PS remoting only if you want to receive commands from a remote machine.
3. **Why I don't see my profile getting loaded when I create a remote session?**
   A: Good question. PowerShell Profiles are not loaded automatically when you create a remote session. You have to manually run the profile script. Or you can create a custom startup script -- as shown in Chapter 7 – to load the profile.ps1, and use –StartupScript parameter to run the script every time you create a remote session.
4. **Is there a place where a comprehensive list of FAQ is provided?**
   A: Yes. Microsoft's TechNet has a complete article on this. Refer to http://technet.microsoft.com/en-us/library/dd315359.aspx
5. **Are there any product specific remoting requirements for products such as SharePoint, Exchange, and etc?**
   Yes. For **SharePoint 2010** remoting, check Zach Rosenfield's blog post: http://sharepoint.microsoft.com/blogs/zach/Lists/Posts/Post.aspx?ID=45
   For **Exchange 2010** remoting, check Mike's blog post: http://www.mikepfeiffer.net/2010/02/managing-exchange-2010-with-remote-powershell/
   For **Lync Server 2010**, check this TechNet blog post: http://blogs.technet.com/b/csps/archive/2010/08/03/scriptremotedesktopicon.aspx
6. **How do I enable unencrypted traffic in a remoting session?**
   You can do so by changing WSMan attributes on the client end. At an elevated PowerShell console,
   Set-Item WSMan:\localHost\Client\AllowUnencrypted –Value $true

Appendix B: Enable PowerShell remoting using group policy

Thanks to Jan Egil Ring for contributing this section to the eBook. His blog was probably the first ever to post detailed steps on how to enable remoting using group policy.

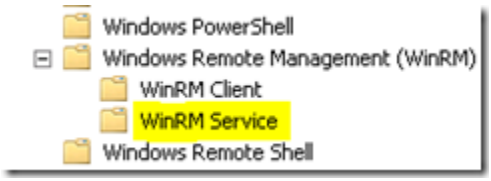
This section will show you how PowerShell remoting can be enabled for Windows Vista, Windows Server 2008 and above. For Windows XP and Windows Server 2003, running Enable-PSRemoting in a PowerShell startup script would be the best approach.

Group Policy Configuration

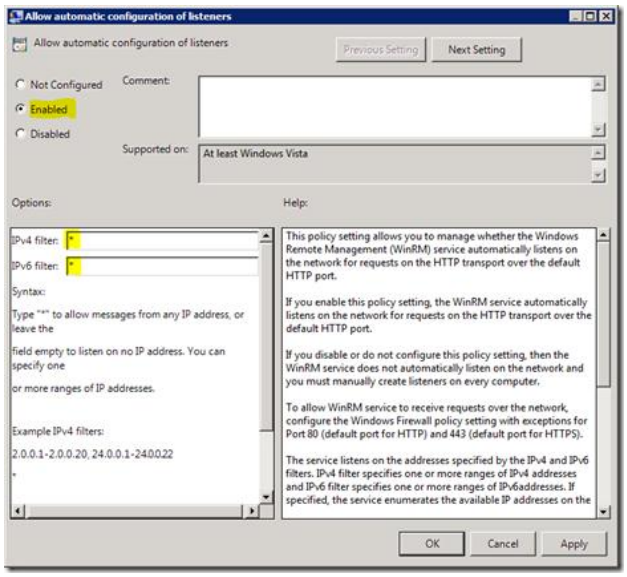
Open the Group Policy Management Console from a domain-joined Windows 7 or Windows Server 2008 R2 computer.

Create or use an existing Group Policy Object, open it, and navigate to Computer Configuration->Policies->Administrative templates->Windows Components

Here you will find the available Group Policy settings for Windows PowerShell, WinRM and Windows Remote Shell:



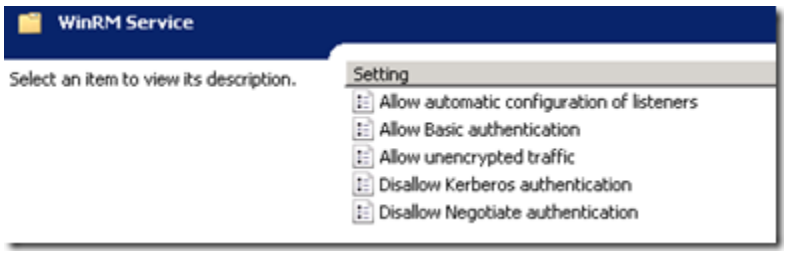
To enable PowerShell Remoting, the only setting we need to configure are found under “WinRM Service”, named “Allow automatic configuration of listeners”:

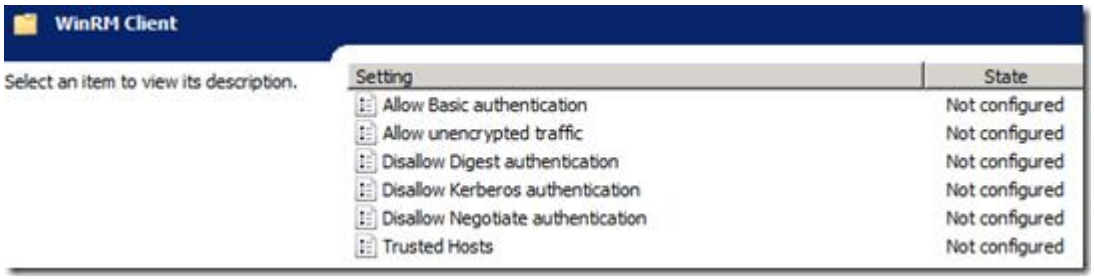


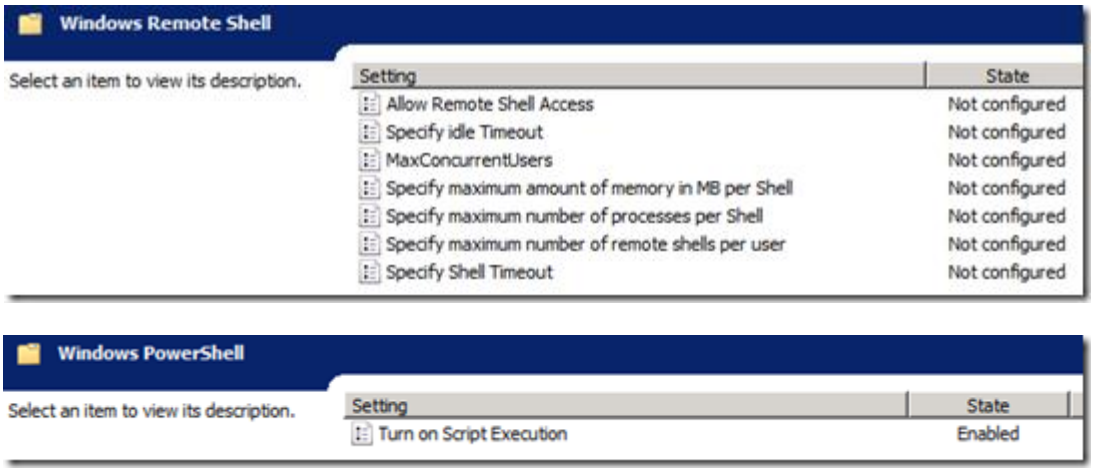
Enable this policy, and configure the IPv4 and IPv6 addresses to listen on. To configure WinRM to listen on all addresses, simply use *.

In addition, the WinRM service is by default not started on Windows client operating systems. To configure the WinRM service to start automatically, navigate to Computer Configuration\Policies\Windows Settings\Security Settings\System Services\Windows Remote Management, double-click on Windows Remote Management and configure the service startup mode to “Automatic”:

No other settings need to be configured, however, I`ve provided screenshots of the other settings so you can see what`s available:



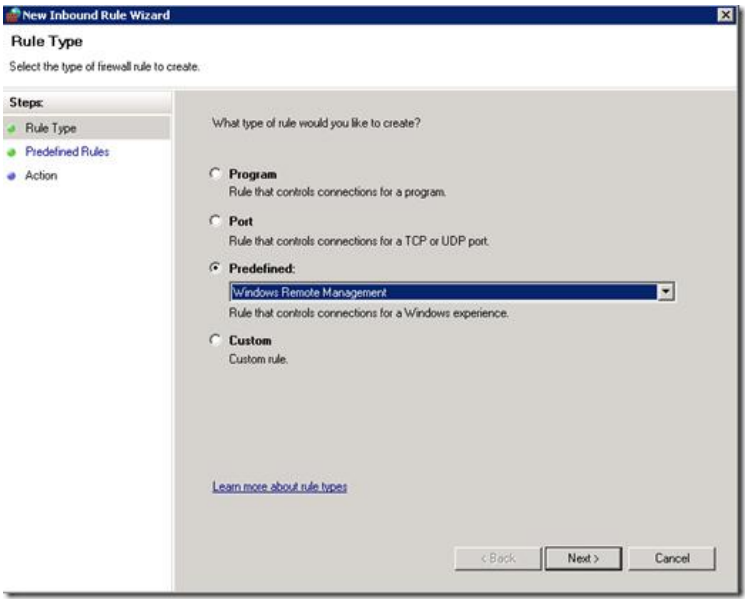




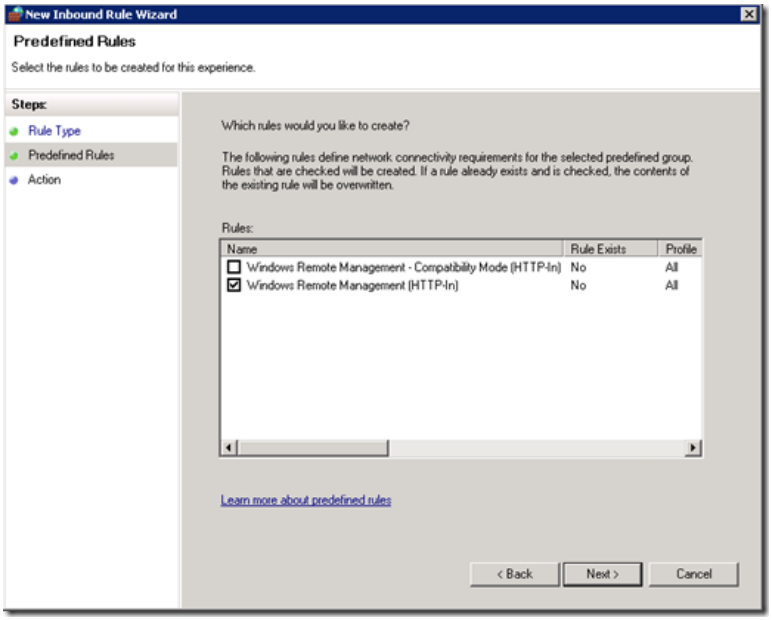
There is one more thing to configure though; the Windows Firewall.

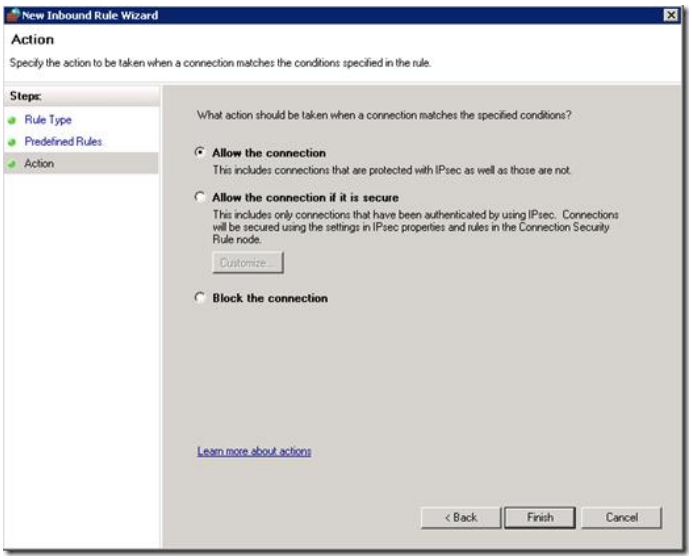
You need to create a new Inbound Rule under Computer Configuration->Policies->Windows Settings->Windows Firewall with Advanced Security->Windows Firewall with Advanced Security->Inbound Rules:

The WinRM port numbers are predefined as “Windows Remote Management”:

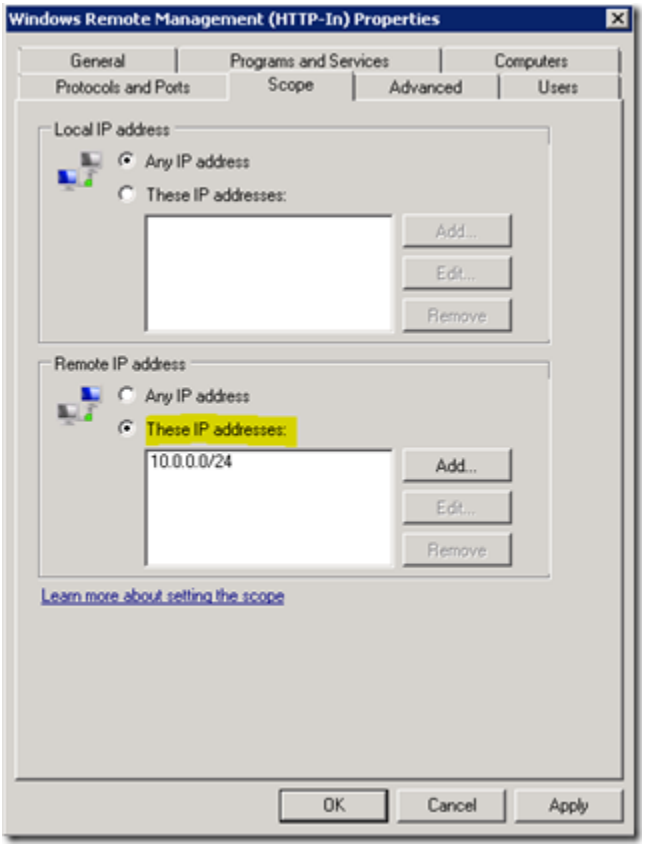


With WinRM 2.0, the default http listener port changed from TCP 80 to TCP 5985. The old port number are a part of the predefined scope for compatibility reasons, and may be excluded if you don`t have any legacy WinRM 1.1 listeners.





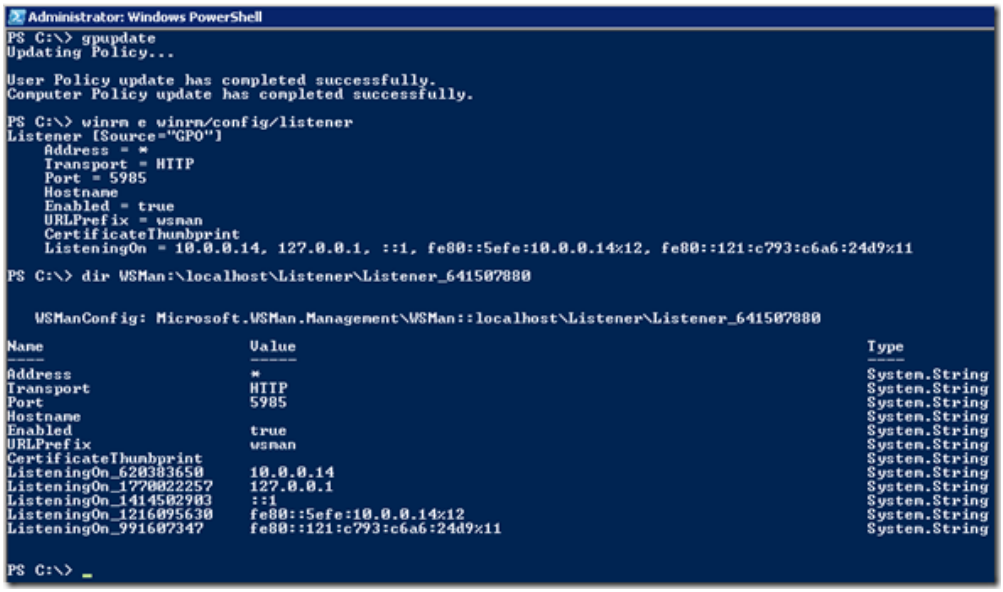
When the rule are created, you may choose to make further restrictions, i.e. to only allow the IP addresses of your management subnet, or perhaps some specific user groups:



Now that the firewall rules are configured, we are done with the minimal configuration to enable PowerShell Remoting using Group Policy.

Name	Group	Profile	Enabled	Action	Override	Program	Local Address	Remote Address	Protocol	Local Port
Windows Remote Management (HTTP-In)	Windows Remote Management	All	Yes	Allow	No	System	Any	10.0.0.0/24, ...	TCP	5985

On a computer affected by the newly configured Group Policy Object, run gpupdate and see if the settings were applied:



As you can see, the listener indicates “Source*”GPO”, meaning it was configured from a Group Policy Object.

When the GPO has been applied to all the affected computers you are ready to test the configuration.

Here is a sample usage of PowerShell Remoting combined with the Active Directory-module for Windows PowerShell:

```
1 Import-Module ActiveDirectory
2 $computers = Get-ADComputer -Filter * -SearchBase "OU=Domain Controllers,DC=domain,DC=local"
3 foreach ($computer in $computers) {
4 Invoke-Command -ComputerName $computer.name -ScriptBlock {
5 Get-Service Netlogon | Select-Object Displayname,Status
6 }
7 }
```

You can also use the Test-PSremoting (http://www.leeholmes.com/blog/2009/11/20/testing-for-powershell-remoting-test-psremoting/) script by Lee Holmes to verify that PS remoting is enabled on remote systems.