3

Exploring PowerShell Remote Management Technologies and PowerShell Remoting

As one of the main purposes of PowerShell is automating administration tasks, **PowerShell remoting** (**PSRemoting**) plays a big part in administrating multiple computers at the same time: using only a single command, you can run the same command line on hundreds of computers.

But similar to when you work with individual computers, PSRemoting is only as secure as your configuration: if you don't lock the door of your house, burglars can break into it.

And that's the same case for computers, as well as for PSRemoting: if you don't harden your configuration and use insecure settings, attackers can leverage that and use your computers against you.

In this chapter, you will not only learn the basics of PSRemoting and how to enable and configure it – you will also discover the best practices for maintaining a secure PSRemoting configuration. While PSRemoting is inherently secure, there are still measures you can take to ensure that your configuration remains secure. We will explore these measures in detail to help you keep your PSRemoting setup secure.

We will also see what PSRemoting network traffic looks like, depending on what authentication protocol is used. Lastly, you will learn how to configure it, what configurations to avoid, and how to use PSRemoting to execute commands.

In this chapter, you will learn about the following topics:

- Working remotely with PowerShell
- Enabling PowerShell remoting
- PowerShell endpoints (session configurations)
- PowerShell remoting authentication and security considerations
- Executing commands using PowerShell remoting
- Working with PowerShell remoting
- PowerShell remoting best practices

Technical requirements

The following are the technical requirements for this chapter:

- PowerShell 7.3 and above
- Visual Studio Code
- Wireshark
- A test lab with a domain controller and one or more test machines
- Access to the GitHub repository for `Chapter03`: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter03

Working remotely with PowerShell

PowerShell was designed to automate administration tasks and simplify the lives of system administrators. Remote management was a part of this plan from the very beginning, as outlined by Jeffrey Snover in the Monad Manifesto from 2002: https://www.jsnover.com/blog/2011/10/01/monad-manifesto/. However, to ship version 1.0 promptly, some features, including PSRemoting, were not included until later versions. PSRemoting was officially introduced in version 2.0 and further improved in version 3.0.

It quickly became one of the most important core functionalities and nowadays supports many other functions within PowerShell, such as workflows.

While PSRemoting can work with a variety of authentication methods, the default protocol for domain authentication is Kerberos. This is the most secure and commonly used method of authentication in Active Directory environments, which is where most people using PSRemoting are likely to be operating. So, when Kerberos is not available, PSRemoting will fall back to NTLM to also support workgroup authentication.

Windows PowerShell supports remoting over different technologies. By default, PSRemoting uses **Windows Remote Management (WinRM)** as its transport protocol. However, it's important to note that WinRM is just one of several protocols that can be used to support remote management in PowerShell. PSRemoting itself is a specific protocol (**PSRP**) that governs the way that PowerShell manages input, output, data streams, object serialization, and more. PSRP can be supported over a variety of transports, including **WS-Management (WS-Man)**, **Secure Shell (SSH)**, **Hyper-V VMBus**, and others. While **Windows Management Instrumentation (WMI)** and **Remote Procedure Call (RPC)** are remote management technologies that can be used with PowerShell, they are not considered part of the PSRemoting protocol.

This difference between those remote management technologies is also reflected in the protocol that's being used:

Remote Connection Method	Protocol Used
PowerShell Remoting via WinRM (default)	WS-Management
WMI	DCOM/RPC
CIM Cmdlets	WS-Management
SSH Remoting	SSH

Table 3.1 – Overview of connection methods and protocols used

PSRemoting is only enabled in Windows Server 2012 R2 and above and only connections from members of the Administrators group are allowed by default. However, PowerShell Core provides support for several remote management protocols, including WMI, Web-Services Management (WS-Management), and SSH remoting. It's important to note that PowerShell Core doesn't support RPC connections.

PowerShell remoting using WinRM

DMTF (formerly known as the **Distributed Management Task Force**) is a non-profit organization that defines open manageability standards, such as the Common Information Model (CIM), and also WS-Management.

WS-Management defines a **Simple Object Access Protocol** (**SOAP**)-based protocol that can be used to manage servers and web services.

Microsoft's implementation of WS-Management is **WinRM**.

As soon as you attempt to establish a PSRemoting connection, the WinRM client sends SOAP messages within the WS-Management protocol over **HTTP** or **HTTPS**.

PSRemoting, when using WinRM, listens on the following ports:

- **HTTP**: **5985**
- **HTTPS**: **5986**

Regardless of whether HTTP or HTTPS is used, PSRemoting traffic is always encrypted after the authentication process – depending on which protocol is used for authentication. You can read more about the different authentication protocols in the *Authentication* section.

On the remote host, the WinRM service runs and is configured to have one or more listeners (HTTP or HTTPS). Each listener waits for incoming HTTP/HTTPS traffic sent through the WS-Management protocol.

Once traffic is received, the WinRM service determines which PowerShell endpoint or application the traffic is meant for and forwards it:

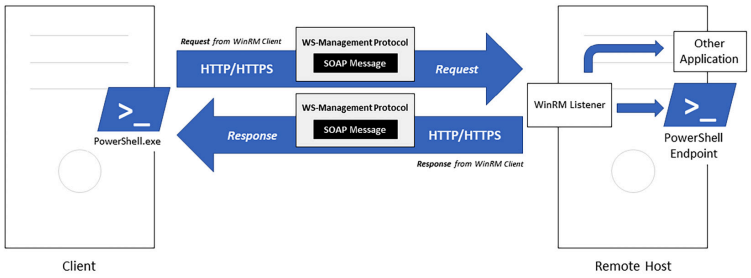

Figure 3.1 – How WinRM and WS-Management are used to connect via PSRemoting

In general, this diagram has been abstracted to simplify your understanding of how WinRM works. **`PowerShell.exe`** is not called; instead, the **`Wsmprovhost.exe`** process is, which runs PSRemoting connections.

As WinRM and WS-Management are the default when establishing remote connections, this chapter will mostly focus on those technologies. But for completeness, I will shortly introduce all other possible remoting technologies in this section.

If you would like to learn about WinRM and WS-Management in more depth, I recommend visiting the following sources:

- **https://docs.microsoft.com/en-us/windows/win32/winrm/windows-remote-management-architecture**
- **https://github.com/devops-collective-inc/secrets-of-powershell-remoting**

Windows Management Instrumentation (WMI) and Common Information Model (CIM)

WMI is Microsoft's implementation of CIM, an open standard designed by DMTF.

WMI was introduced with Windows NT 4.0 and was included in the Windows operating system starting with Windows 2000. It is still present in all modern systems, including Windows 10 and Windows Server 2019.

CIM defines how IT system elements are represented as objects and how they relate to each other. This should offer a good way to manage IT systems, regardless of the manufacturer or platform.

WMI relies on the **Distributed Component Object Model** (**DCOM**) and RPC, which is the underlying mechanism behind DCOM, to communicate.

DCOM was created to let the **Component Object Model** (**COM**) communicate over the network and is the predecessor of .NET Remoting.

This section will give you only a basic overview of the WMI and CIM cmdlets to fulfill your understanding of the remote management technologies in this chapter. You will learn more about COM, WMI, and CIM in ***Chapter 5***, *PowerShell Is Powerful – System and API Access*.

WMI cmdlets

WMI cmdlets were deprecated starting with PowerShell Core 6 and should not be used in newer versions of PowerShell. However, it's important to note that they are still supported in certain older versions of PowerShell, such as PowerShell 5.1 on Windows 10, and will continue to be supported for the support life of those operating systems. If possible, use the newer CIM cmdlets instead, since they can be used on Windows and non-Windows operating systems.

First, let's have a look at how to work with the deprecated, but still present, WMI cmdlets.

To find all the cmdlets and functions that have the **wmi** string included in their name, leverage the **Get-Command** cmdlet. With the **-CommandType** parameter, you can specify what kind of commands you want to look for. In this example, I am searching for cmdlets and functions:

```
> Get-Command -Name *wmi* -CommandType Cmdlet,Function
CommandType  Name               Version    Source
-----------  ----               -------    ------
Cmdlet       Get-WmiObject      3.1.0.0    Microsoft.PowerShell.Management
Cmdlet       Invoke-WmiMethod   3.1.0.0    Microsoft.PowerShell.Management
Cmdlet       Register-WmiEvent  3.1.0.0    Microsoft.PowerShell.Management
Cmdlet       Remove-WmiObject   3.1.0.0    Microsoft.PowerShell.Management
Cmdlet       Set-WmiInstance    3.1.0.0    Microsoft.PowerShell.Management
```

An example of how to work with WMI is via the **Get-WmiObject** cmdlet. Using this cmdlet, you can query local and remote computers.

You can use the **-List** parameter to retrieve all available WMI classes on your computer:

```
> Get-WmiObject -List
   NameSpace: ROOT\cimv2
Name                  Methods Properties
----                  ------- ----------
CIM_Indication        {}      {CorrelatedIndications, IndicationFilterName, IndicationIde...
CIM_ClassIndication   {}      {ClassDefinition, CorrelatedIndications, IndicationFilterNa...
CIM_ClassDeletion     {}      {ClassDefinition, CorrelatedIndications, IndicationFilterNa...
...
```

Here's an example of how to use **Get-WmiObject** to retrieve information about Windows services on your local computer:

```
> Get-WmiObject -Class Win32_Service
ExitCode  : 0
Name      : AdobeARMservice
ProcessId : 3556
StartMode : Auto
State     : Running
Status    : OK
…
```

Not only can you query your local computer, but you can also query a remote computer by using the **-ComputerName** parameter, followed by the name of the remote computer. The following example shows how to retrieve the same information from the **PSSec-PC02** remote computer:

```
> Get-WmiObject -Class Win32_Service -ComputerName PSSec-PC02
```

The preceding code returns a list of all services that are available on the remote computer.

By using the **-Query** parameter, you can even specify the query that should be run against the CIM database of the specified computer. The following command only retrieves all services with the name **WinRM**:

```
> Get-WmiObject -ComputerName PSSec-PC02 -Query "select * from win32_service where name='WinRM'"
ExitCode  : 0
```

```
Name      : WinRM
ProcessId : 6408
StartMode : Auto
State     : Running
Status    : OK
```

In this example, we run the specified **`select * from win32_service where name='WinRM'`** query remotely on **`PSSec-PC02`**.

Using PowerShell WMI cmdlets, you can also call WMI methods, delete objects, and much more.

DID YOU KNOW?

RPC, on which WMI relies, is no longer supported in PowerShell Core 6. This is due in part to PowerShell's goal of cross-platform compatibility: from PowerShell version 7 and above, RPC is only supported on machines running the Windows operating system.

CIM cmdlets

With PowerShell 3.0, which came with Windows Server 2012 and Windows 8, a new set of cmdlets were introduced to manage objects that were compliant with the CIM and WS-Man standards.

At some point, the WMI cmdlets drifted away from the DMTF standards, which prevented cross-platform management. So, Microsoft moved back to being compliant with the DMTF CIM standards by publishing the new CIM cmdlets.

To find out all CIM-related cmdlets, you can leverage the **`Get-Command`** cmdlet:

```
> Get-Command -Name "*cim*" -CommandType Cmdlet,Function
CommandType     Name                          Version    Source
-----------     ----                          -------    ------
Cmdlet          Get-CimAssociatedInstance     1.0.0.0    CimCmdlets
Cmdlet          Get-CimClass                  1.0.0.0    CimCmdlets
Cmdlet          Get-CimInstance               1.0.0.0    CimCmdlets
Cmdlet          Get-CimSession                1.0.0.0    CimCmdlets
Cmdlet          Invoke-CimMethod              1.0.0.0    CimCmdlets
Cmdlet          New-CimInstance               1.0.0.0    CimCmdlets
Cmdlet          New-CimSession                1.0.0.0    CimCmdlets
Cmdlet          New-CimSessionOption          1.0.0.0    CimCmdlets
Cmdlet          Register-CimIndicationEvent   1.0.0.0    CimCmdlets
Cmdlet          Remove-CimInstance            1.0.0.0    CimCmdlets
Cmdlet          Remove-CimSession             1.0.0.0    CimCmdlets
Cmdlet          Set-CimInstance               1.0.0.0    CimCmdlets
```

In this example, we are looking for all cmdlets and functions that have **`cim`** in their name.

You can find an overview of all the currently available CIM cmdlets to interact with the CIM servers at **https://docs.microsoft.com/de-de/powershell/module/cimcmdlets/**.

Open Management Infrastructure (OMI)

To help with a cross-platform managing approach, Microsoft created the **Open Management Infrastructure** (**OMI**) in 2012 (https://github.com/Microsoft/omi), but it never really became that popular and isn't used broadly anymore. Therefore, Microsoft decided to add support for SSH remoting.

PowerShell remoting using SSH

To enable PSRemoting between Windows and Linux hosts, Microsoft added support for PSRemoting over **SSH** with PowerShell 6.

PSREMOTING VIA SSH REQUIREMENTS

To use PSRemoting via SSH, PowerShell version 6 or above and SSH need to be installed on all computers. Starting from Windows 10 version 1809 and Windows Server 2019, OpenSSH for Windows was integrated into the Windows operating system.

PowerShell remoting on Linux

As a first step, to use PowerShell on Linux, install PowerShell Core by following the steps for your operating system, which you can find in the official PowerShell Core documentation: **https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-linux**.

In my demo lab, I have a Debian 10 server installed. So, the steps may vary, depending on the operating system that is used.

Configure **`/etc/ssh/sshd_config`** with the editor of your choice. In my example, I am using **vi**:

```
> vi /etc/ssh/sshd_config
```

First, add a PowerShell subsystem entry to your configuration:

```
Subsystem powershell /usr/bin/pwsh -sshs -NoLogo
```

In Linux systems, the PowerShell executable is typically located at **`/usr/bin/pwsh`** by default. Please make sure you adjust this part if you installed PowerShell in a different location.

To allow users to log on remotely using SSH, configure **`PasswordAuthentication`** and/or **`PubkeyAuthentication`**:

- If you want to allow authentication using a username and a password, set **`PasswordAuthentication`** to **`yes`**:

  ```
  PasswordAuthentication yes
  ```

- If you want to enable a more secure method, set **`PubkeyAuthentication`** to **`yes`**:

  ```
  PubkeyAuthentication yes
  ```

`PubkeyAuthentication`, which stands for **public key authentication**, is a method of authentication that relies on a generated key pair: a private

and a public key is generated. While the **private key** is kept safe on the user's computer, the **public key** is entered on a remote server.

When the user authenticates using this private key, the server can verify the user's identity using their public key. A public key can only be used to verify the authenticity of the private key or to encrypt data that only the private key can encrypt.

Using public key authentication for remote access not only protects against the risk of password attacks such as brute-force and dictionary attacks but also offers an additional layer of security in case the server gets compromised. In such cases, only the public key can be extracted while the private key remains safe. As the public key alone is not enough to authenticate, this method provides better security than using a username and password, as passwords can be extracted and reused if the server is compromised.

You can learn how to generate a key pair using the **ssh-keygen** tool at https://www.ssh.com/ssh/keygen/.

If you are interested in how public key authentication works, you can read more about it on the official SSH website: **https://www.ssh.com/ssh/public-key-authentication**.

Of course, both authentication mechanisms can be configured at the same time, but if you use `PubkeyAuthentication` and no other user connects using their username and password, you should use `PubkeyAuthentication` only:

```
PasswordAuthentication no
PubkeyAuthentication yes
```

If you want to learn more about the different options of the `sshd` configuration file, I highly recommend that you look at the **man pages**: https://manpages.debian.org/jessie/openssh-server/sshd_config.5.en.html.

MAN PAGES

Man *stands for* **manual**. *Man pages are used to get more information about a Linux/UNIX command or configuration file and can be compared to the Help system in PowerShell.*

Restart the `ssh` service:

```
> /etc/init.d/ssh restart
```

The updated configuration is loaded into memory to activate the changes.

PowerShell remoting on macOS

To enable PSRemoting over SSH to manage macOS systems, the steps are quite similar to those when enabling PSRemoting on a Linux system: the biggest difference is that the configuration files are in a different location.

First, you need to install PowerShell Core on the macOS systems that you want to manage remotely: **https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-macos**.

Edit the **ssh** configuration:

```
> vi /private/etc/ssh/sshd_config
```

Create a subsystem entry for PowerShell:

```
Subsystem powershell /usr/local/bin/pwsh -sshs -NoLogo
```

Then, define what kind of authentication you want to configure for this machine:

- Username and password:
  ```
  PasswordAuthentication yes
  ```
- Public key authentication:
  ```
  PubkeyAuthentication yes
  ```

To learn more about the options that can be configured in the **sshd** configuration, have a look at the *PowerShell remoting on Linux* section that we covered previously.

Restart the **ssh** service to load the new configuration:

```
> sudo launchctl stop com.openssh.sshd
> sudo launchctl start com.openssh.sshd
```

The service will restart and the new configuration will be active.

PowerShell remoting via SSH on Windows

Of course, it is also possible to manage Windows systems via SSH, but in this book, I will use PSRemoting via WinRM in all of my examples as this is the default setting on Windows systems.

However, if you want to enable PSRemoting via SSH on your Windows systems, make sure you install OpenSSH and follow the instructions on how to set up PSRemoting over SSH on Windows:

- **https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh_overview**
- **https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/ssh-remoting-in-powershell-core?view=powershell-7.1#set-up-on-a-windows-computer**

DID YOU KNOW?

PSRemoting via SSH does not support remote endpoint configuration, nor **Just Enough Administration** *(***JEA***).*

Enabling PowerShell remoting

There are different ways to enable PSRemoting for your system(s). If you only work with a few machines in your lab, you might want to enable it manually. But as soon as you want to enable PSRemoting in a big environment, you might want to enable and configure PSRemoting centrally. In this section, we will have a look at both methods. The following table provides an overview of which method takes which configuration actions:

Action	Enable-PSRemoting	Group Policy	Manual Configuration
Set the WinRM to Auto-Start	Yes	Yes	Yes
Configure HTTP Listener	Yes	Yes *(No Custom Listeners)*	Yes
Configure HTTPS Listener	No	No	Yes
Configure Endpoints	Yes	No	Yes
Configure Firewall	Yes	Yes	Yes

Table 3.2 – Enabling PSRemoting – different methods

Please note that the **`Enable-PSRemoting`** method is a subpart of the manual configuration; to configure HTTP and HTTPS listeners, additional steps must be taken. Let's explore what is needed to manually configure PSRemoting, which could be useful in a test scenario, for example.

Enabling PowerShell remoting manually

If you want to enable PSRemoting on a single machine, this can be done manually by using the **`Enable-PSRemoting`** command on an elevated shell:

```
> Enable-PSRemoting
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.
WinRM has been updated for remote management.
WinRM firewall exception enabled.
Configured LocalAccountTokenFilterPolicy to grant administrative rights remotely to local users.
```

In this example, the command ran successfully, so PSRemoting was enabled on this machine.

If you're wondering about the difference between **`Enable-PSRemoting`** and **`winrm quickconfig`**, the truth is that there is not much difference technically. **`Enable-PSRemoting`** already incorporates all the actions performed by **`winrm quickconfig`**, but with additional environment changes specific to Windows PowerShell. So, to put it simply, running **`Enable-PSRemoting`** is sufficient, and you can skip running **`winrm quickconfig`**.

Set-WSManQuickConfig error message

Depending on your network configuration, an error message may be shown if you try to enable PSRemoting manually:

```
WinRM firewall exception will not work since one of the network connection types on this machine i
```

This error message was generated by the **`Set-WSManQuickConfig`** command, which is called during the process of enabling PSRemoting.

This message is shown if one network connection is set to public because, by default, PSRemoting is not allowed on networks that were defined as public networks:

```
> Get-NetConnectionProfile
Name             : Network 1
InterfaceAlias   : Ethernet
InterfaceIndex   : 4
NetworkCategory  : Public
IPv4Connectivity : Internet
IPv6Connectivity : NoTraffic
```

To avoid this error, there are two options:

- Configure the network profile as a private network.
- Enforce **`Enable-PSRemoting`** so that the network profile check is skipped.

If you are certain that the network profile is not a public one and instead a network that you trust, you can configure it as a private network:

```
> Set-NetConnectionProfile -NetworkCategory Private
```

If you don't want to configure the network as a trusted, private network, you can enforce skipping the network profile check by adding the **`-SkipNetworkProfileCheck`** parameter:

```
> Enable-PSRemoting -SkipNetworkProfileCheck
```

Having PSRemoting enabled on public network-connected computers puts your computer at significant risk, so be careful.

Checking your WinRM configuration

After enabling PSRemoting and WinRM, you might want to check the current WinRM configuration. You can achieve this using **`winrm get winrm/config`**:

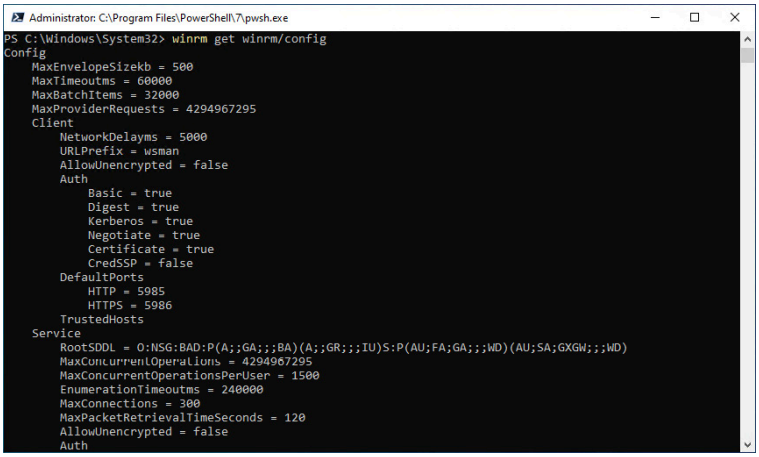

Figure 3.2 – Verifying your local WinRM configuration

You can find all the configured options in the displayed output. The **`winrm get winrm/config`** command provides a summary of the WinRM configuration settings.

To change your local WinRM configuration, you can use the **`set`** option:

```
> winrm set winrm/config/service '@{AllowUnencrypted="false"}'
```

Alternatively, you can use the **`wsman:\`** PowerShell drive to access and modify specific items in the configuration. Using the **`wsman:\`** provider allows you to access and modify specific items of the WinRM configuration in a more intuitive and cmdlet-like way, with the added benefit of built-in help and documentation.

To change your local WinRM configuration, you can use the **`Set-Item`** cmdlet with the **`wsman:\`** provider to access and modify the WinRM configuration items. For example, to disable the use of unencrypted traffic, you can run the following command:

```
> Set-Item wsman:\localhost\Service\AllowUnencrypted -Value $false
```

In this example, we are configuring the WinRM service to *not* allow unencrypted connections. You can use a similar syntax to also configure other WinRM options – just make sure you provide the entire path to the setting in the tree, as well as the option and the value.

Trusted hosts

If you are connecting to a machine that is not domain-joined, which might be the reason why you configure it manually, Kerberos authentication is not an option and the NTLM protocol should be used for authentication instead.

In this case, you need to configure the remote machine to be considered a trusted host in **WS-Man** on your local device; otherwise, the connection will fail.

To configure **`TrustedHosts`** for a remote host, you can use the **`Set-Item`** cmdlet, along with the **`wsman:\localhost\client\TrustedHosts`** path. By default, this value is empty, so you need to add the IP address or domain name of the remote host. To add a new value without replacing the existing ones, use the **`-Concatenate`** switch, as shown here:

```
> Set-Item wsman:\localhost\client\TrustedHosts -Value 172.29.0.12 -Concatenate -Force
```

This will append the specified IP address to the existing list of **`TrustedHosts`**.

To verify that your changes were applied, you can use the **`Get-Item`** cmdlet to display the current **`TrustedHosts`** configuration:

```
> Get-Item wsman:\localhost\client\TrustedHosts
   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client
Type            Name             SourceOfValue   Value
----            ----             -------------   -----
System.String   TrustedHosts                     172.29.0.12
```

The preceding example shows that the host with an IP address of **`172.29.0.12`** has been configured as a trusted host on the local machine.

It is also a good practice to audit the **`TrustedHosts`** list to detect any unauthorized changes. This can help in detecting tampering attempts on your system.

Connecting via HTTPS

Optionally, you can also configure a certificate to encrypt the traffic over **HTTPS**. To ensure secure PSRemoting, it is recommended that you configure a certificate to encrypt the traffic over HTTPS, especially in scenarios where Kerberos is not available for server identity verification. Although PSRemoting traffic is encrypted by default, encryption can be removed, and basic authentication can be enforced easily (see the *PowerShell remoting authentication and security considerations* section). Configuring a certificate adds another layer of security to your environment.

Therefore, to provide an extra layer of security, it can make sense to issue a certificate and enable **WinRM via SSL**.

If you haven't purchased a publicly signed SSL certificate from a valid **certificate authority** (**CA**), you can create a **self-signed certificate** to get started. However, if you're using this for workgroup remoting, you can also use an **internal CA**. This can provide additional security and trust since you have a trusted source within the organization sign the certificate.

This section only covers how to issue and configure a self-signed certificate. So, make sure you adjust the steps if you are using a publicly signed certificate or an internal CA.

First, let's get a self-signed certificate! This step is very easy if you are working on Windows Server 2012 and above – you can leverage the **`New-SelfSignedCertificate`** cmdlet:

```
> $Cert = New-SelfSignedCertificate -CertstoreLocation Cert:\LocalMachine\My -DnsName "PSSec-PC01"
> Export-Certificate -Cert $Cert -FilePath C:\tmp\cert
```

Make sure that the value provided via the **`-DnsName`** parameter matches the hostname and that a matching DNS record exists in your DNS server.

Add an HTTPS listener:

```
> New-Item -Path WSMan:\LocalHost\Listener -Transport HTTPS -Address * -CertificateThumbPrint $Cer
```

Finally, make sure you add an exception for the firewall. The default port for WinRM over HTTPS is **`5986`**:

```
> New-NetFirewallRule -DisplayName "Windows Remote Management (HTTPS-In)" -Name "Windows Remote Ma
```

To clarify, it's important to note that using the **`-Profile Any`** option opens WinRM to public or unidentified networks. If you're not in a test environment, make sure you use the appropriate profile options, such as **`Domain`**, **`Private`**, or **`Public`**.

If you want to ensure that only HTTPS is used, remove WinRM's HTTP listener:

```
> Get-ChildItem WSMan:\Localhost\listener | Where -Property Keys -eq "Transport=HTTP" | Remove-Ite
```

Additionally, you may want to check and remove any existing firewall exceptions for HTTP traffic that were configured. This step is not necessary if you did not configure any exceptions previously.

In some cases, you may want to move the WinRM listener to a different port. This can be useful if your firewall setup does not allow port **`5986`** or if you want to use a non-standard port for security reasons. To move the WinRM listener to a different port, use the **`Set-Item`** cmdlet:

```
> Set-Item WSMan:\Localhost\listener\<ListenerName>\port -Value <PortNumber>
```

Replace **`<ListenerName>`** with the name of the listener that you want to edit and replace **`<PortNumber>`** with the port number that you want to configure.

Next, we'll import our certificate. However, before doing so, it's important to understand that certificates generated through tools such as **`New-SelfSignedCertificate`** already have usage restrictions built into them to ensure they are only valid for client and server authentication. If you're using a certificate generated through another tool (for example, an internal PKI), it's important to make sure that it also has these usage restrictions. Additionally, ensure that the root certificate is protected properly since attackers can use it to forge SSL certificates for trusted websites.

Once you have the appropriate certificate, copy it to a secure location on the computer from where you want to connect to the remote machine (such as **`C:\tmp\cert`** in our example), and then import it into the local certificate store:

```
> Import-Certificate -Filepath "C:\tmp\cert" -CertStoreLocation "Cert:\LocalMachine\Root"
```

Specify the credentials that you want to use to log in and enter your session. The **`-UseSSL`** parameter indicates that your connection will be encrypted using SSL:

```
> $cred = Get-Credential
> Enter-PSSession -ComputerName PSSec-PC01 -UseSSL -Credential $cred
```

Of course, you still have to enter credentials to sign in to the machine remotely. The certificate only guarantees the authenticity of the remote computer and helps establish the encrypted connection.

Configuring PowerShell Remoting via Group Policy

When working with multiple servers, you may not want to enable PSRemoting manually on each machine, so Group Policy is the tool of your choice.

Using Group Policy, you can configure multiple machines using a single **Group Policy Object** (**GPO**).

To get started, create a new GPO: open **Group Policy Management**, right-click on the **Organizational Unit** (**OU**) in which you want to create the new GPO, and select **Create a GPO in this domain, and Link it here...**.

GPO is only a tool to configure your machines – it doesn't start services. Therefore, you still need to find a solution to reboot all configured servers or start the WinRM service on all servers.

If you want to enable PSRemoting remotely, Lee Holmes has written a great script that leverages WMI connections (which most systems support): **http://www.powershellcookbook.com/recipe/SQOK/program-remotely-enable-powershell-remoting**.

Allowing WinRM

In the newly created GPO, navigate to **Computer Configuration** | **Policies** | **Administrative Templates** | **Windows Components** | **Windows Remote Management** | **WinRM Service** and set the **Allow remote server management through WinRM** policy to **Enabled**.

In this policy, you can define the IPv4 and IPv6 filters. If you don't use a protocol (for example, IPv6), then leave it empty so that users can't connect to WinRM using this particular protocol.

To allow connections, you can use the wildcard character, *****, an IP, or an IP range.

When working with customers or in my demo labs, I learned that the most common reason for errors occurring regarding why WinRM did not work was using an IP or an IP range when configuring this setting.

Therefore, nowadays, I use the wildcard character, *****, *but only* in combination with a **firewall IP restriction**, to secure my setup. We will configure the firewall IP restriction later in this section (see *Creating a firewall rule*):

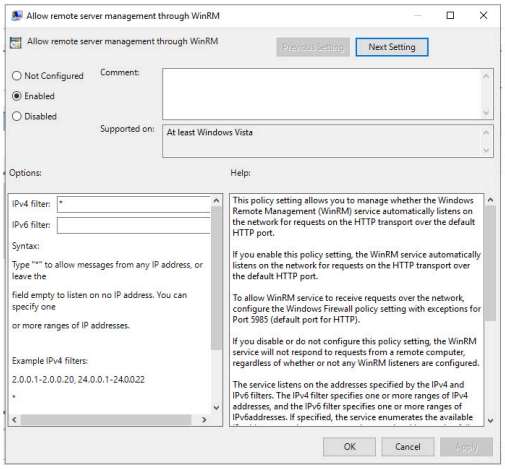

Figure 3.3 – Configuring Allow remote server management through WinRM

CAUTION!

Only use the wildcard (*) configuration if you wish to restrict via a firewall rule that remote IPs are allowed to connect to.

Configuring the WinRM service to start automatically

To configure the WinRM service so that it starts automatically, follow these steps:

1. Use the same GPO and navigate to **Computer Configuration** | **Policies** | **Windows Settings** | **Security Settings** | **System Services**.
2. Select and configure the **Windows Remote Management (WS Management)** setting.
3. A new window will open. Check the **Define this policy setting** option and set the service startup mode to **Automatic**.
4. Confirm your configuration by clicking **OK**:

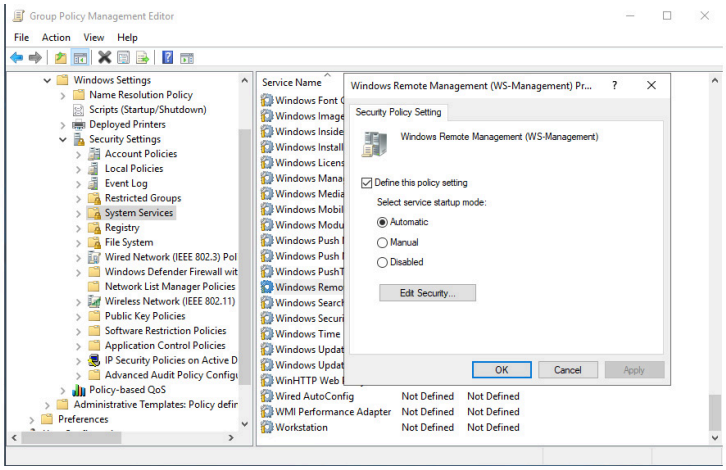

Figure 3.4 – Configuring the Windows Remote Management service so that it starts automatically

NOTE

This setting only configures the service to start automatically, which usually happens when your computer starts. It does not start the service for you, so make sure that you reboot your computer (or start the service manually) so that the WinRM service starts automatically.

Creating a firewall rule

To configure the settings of the firewall, follow these steps:

1. Navigate to **Computer Configuration** | **Policies** | **Windows Settings** | **Security Settings** | **Windows Defender Firewall with Advanced Security** | **Windows Defender Firewall with Advanced Security** | **Inbound Rules**.
2. Create a new inbound rule using the wizard.
3. Check the **Predefined** option and select **Windows Remote Management**:

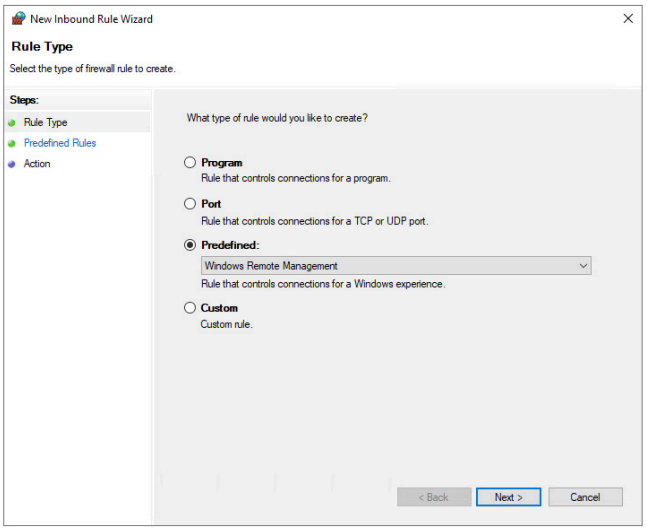

Figure 3.5 – Creating a predefined Windows Remote Management firewall rule

4. Click **Next** and remove the **Public** firewall profile by deselecting the option shown in the following screenshot:

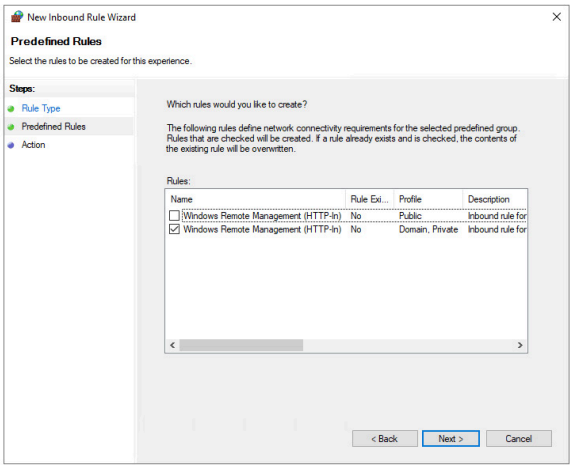

Figure 3.6 – Deselecting the public network profile

5. Finally, select **Allow the connection** before confirming your configuration by clicking the **Finish** button:

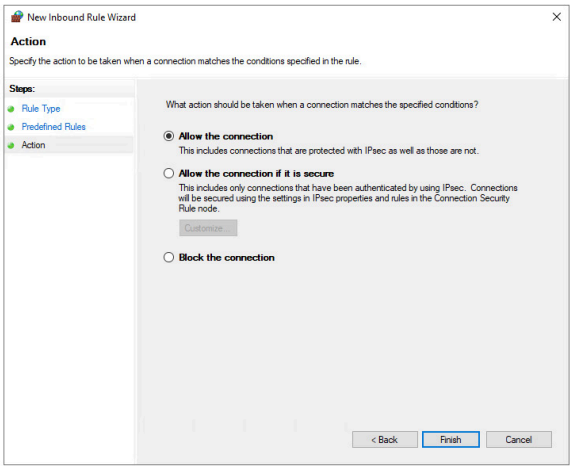

Figure 3.7 – Allow the connection

The new rule will be created, and shown in your GPO:

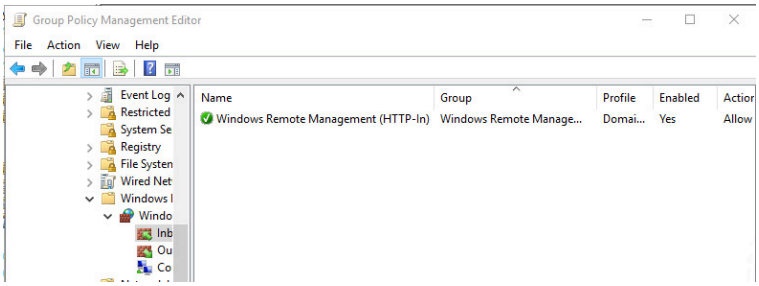

Figure 3.8 – Displaying the new inbound firewall rule

6. Before exiting the GPO configuration, make sure you open your newly created firewall rule once again by double-clicking it. The **Windows Remote Management (HTTP-In) Properties** window will open.
7. Optional: if your machines reside in the same domain, navigate to the **Advanced** tab and deselect the **Private** profile to make sure that a remote connection using WinRM is only allowed within the **Domain** network profile:

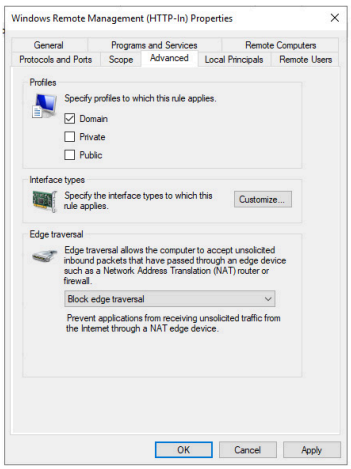

Figure 3.9 – Only allowing WinRM within the Domain network profile

8. Then, navigate to the **Scope** tab and add all remote IP addresses from which it should be allowed to access the computer remotely. For instance, if you have a management subnet on your network, you can add the IP addresses within that subnet to the list:

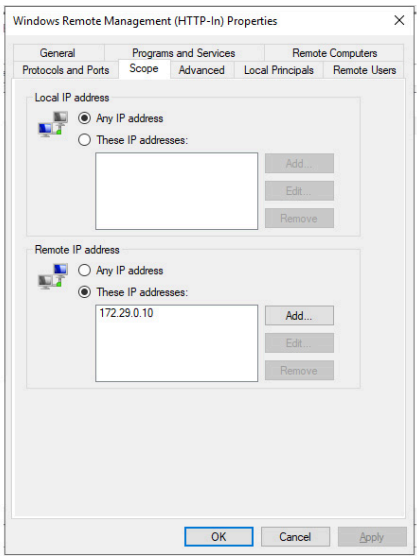

Figure 3.10 – Configuring which remote IP addresses are allowed to connect

In the best case, allow only a hardened, secure management system to manage systems via PSRemoting.

Use the clean source principle to build the management system and use the recommended privileged access model to access it:

- **https://learn.microsoft.com/en-us/security/privileged-access-workstations/privileged-access-success-criteria#clean-source-principle**
- **https://learn.microsoft.com/en-us/security/privileged-access-workstations/privileged-access-access-model**

PowerShell endpoints (session configurations)

In this chapter, you might have read the term **endpoint** several times.

If we are talking about endpoints, we are not talking about one computer: PSRemoting is designed to work with multiple endpoints on a computer.

But what exactly is an endpoint?

When we are talking about PowerShell endpoints, *each endpoint is a session configuration*, which you can configure to offer certain services or which you can also restrict.

So, every time we run `Invoke-Command` or enter a PowerShell session, we are connecting to an endpoint (also known as a remote session configuration).

Sessions that offer fewer cmdlets, functions, and features, as those that are usually available if no restrictions are in place, are called **constrained endpoints**.

Before we enable PSRemoting, no endpoint will have been configured on the computer.

You can see all the available session configurations by running the **`Get-PSSessionConfiguration`** command:

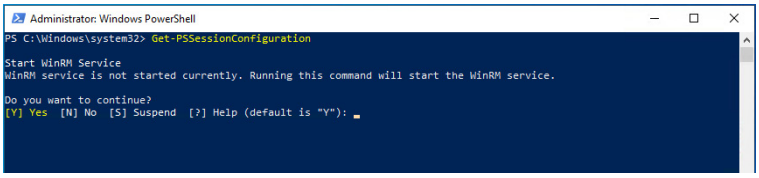

Figure 3.11 – No endpoint is shown when PSRemoting is not enabled

When PSRemoting is not enabled on a computer, no endpoint will be shown. This is because the WinRM service, which is responsible for PSRemoting, is not started by default. However, once the WinRM service is started, the endpoints are already configured and ready to use, but not exposed and cannot be connected to until PSRemoting is enabled.

Enabling PSRemoting using **`Enable-PSRemoting`**, as we did in the previous section, creates all default session configurations, which are necessary to connect to this endpoint via PSRemoting:

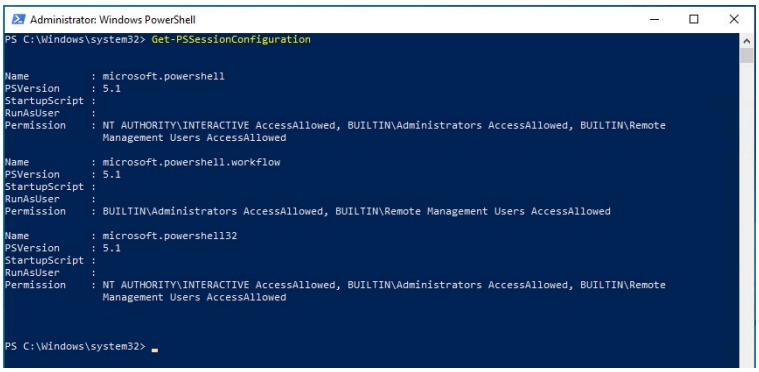

Figure 3.12 – After enabling PSRemoting, we can see all the prepopulated endpoints

Typically, in Windows PowerShell 3.0 and above, there are three default preconfigured endpoints on client systems:

- **`microsoft.powershell`**: This is the standard endpoint and is used for PSRemoting connections if not specified otherwise
- **`microsoft.powershell32`**: This is a 32-bit endpoint that's optional if you're running a 64-bit operating system
- **`microsoft.powershell.workflow`**: This endpoint is for PowerShell workflows – h**ttps://docs.microsoft.com/en-us/system-center/sma/overview-powershell-workflows?view=sc-sma-2019**

On *server systems*, there's typically a fourth session configuration that's predefined:

- **`microsoft.windows.servermanagerworkflows`**: This endpoint is for Server Manager workflows – https://docs.microsoft.com/en-us/windows-server/administration/server-manager/server-manager

Every computer will show different default endpoints. In the preceding example, I ran the command on a Windows 10 client, which will show fewer endpoints than, for example, Windows Server 2019.

Connecting to a specified endpoint

By default, the **`microsoft.powershell`** endpoint is used for all PSRemoting connections. But if you want to connect to another specified endpoint, you can do this by using the **`-ConfigurationName`** parameter:

```
> Enter-PSSession -ComputerName PSSec-PC01 -ConfigurationName 'microsoft.powershell32'
```

The specified configuration can be either the name of another default or a custom endpoint.

Creating a custom endpoint – a peek into JEA

Creating a custom endpoint (also known as **Just Enough Administration** or **JEA**) allows you to define a restricted administrative environment for delegated administration. With JEA, you can define a set of approved commands and parameters that are allowed to be executed on specific machines by specific users. This enables you to give users just enough permissions to perform their job duties, without granting them full administrative access. It is a great way to secure your remote connections:

- You can restrict the session so that only predefined commands will be run.
- You can enable transcription so that every command that is executed in this session is logged.
- You can specify a security descriptor (SDDL) to determine who is allowed to connect and who isn't.
- You can configure scripts and modules that will be automatically loaded as soon as the connection to this endpoint is established.
- You can even specify that another account is used to run your commands in this session on the endpoint.

To create and activate an endpoint, two steps need to be followed:

1. Creating a session configuration file
2. Registering the session as a new endpoint

Creating a session configuration file

Using **`New-PSSessionConfigurationFile`**, you can create an empty skeleton session configuration file. You need to specify the path where the configuration file will be saved, so the **`-Path`** parameter is mandatory. A session configuration file ends with the **`.pssc`** filename extension, so make sure you name the file accordingly:

```
> New-PSSessionConfigurationFile -Path <Path:\To\Your\SessionConfigurationFile.pssc>
```

Have a look at the official documentation for more information: **https://docs.microsoft.com/en-**

us/powershell/module/microsoft.powershell.core/new-pssessionconfigurationfile.

You can either generate an empty session configuration file and populate it later using an editor or you can use the **New-PSSessionConfigurationFile** parameters to directly generate the file with all its defined configuration options:

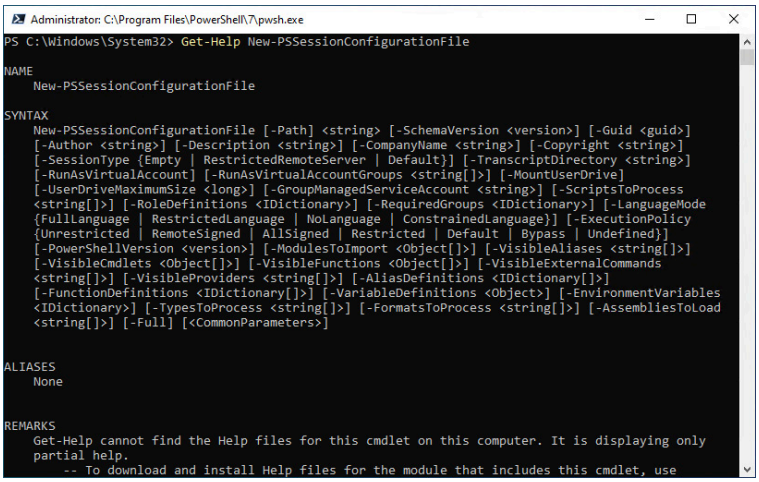

Figure 3.13 – New-PSSessionConfigurationFile parameters

For this example, we will create a session configuration file for a **RestrictedRemoteServer** session:

```
> New-PSSessionConfigurationFile -SessionType RestrictedRemoteServer -Path .\PSSessionConfig.pssc
```

By using **-SessionType RestrictedRemoteServer**, only the most important commands are being imported into this session, such as **Exit-PSSession**, **Get-Command**, **Get-FormatData**, **Get-Help**, **Measure-Object**, **Out-Default**, and **Select-Object**. If you want to allow other commands in this session, they need to be configured in the role capability file, which we will discuss in detail in ***Chapter 10***, *Language Modes and Just Enough Administration (JEA)*.

Registering the session as a new endpoint

After creating the session configuration file, you must register it as an endpoint by utilizing the **Register-PSSessionConfiguration** command.

When utilizing the mandatory **-Name** parameter, make sure you only specify the name of the session configuration file, without including the filename extension:

```
> Register-PSSessionConfiguration -Name PSSessionConfig
WARNING: Register-PSSessionConfiguration may need to restart the WinRM service if a configuration
All WinRM sessions connected to Windows PowerShell session configurations, such as Microsoft.Power
   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin
Type            Keys                                Name
----            ----                                ----
Container       {Name=PSSessionConfig}              PSSessionConfig
```

The session configuration will be registered, and a new endpoint will be created. Sometimes, it might be necessary to restart the WinRM service after registering an endpoint:

```
> Get-PSSessionConfiguration -Name PSSessionConfig
Name          : PSSessionConfig
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed, BUILTIN\Administrators AccessAllowed, BUIL
```

Using **`Get-PSSessionConfiguration`**, you can verify that the endpoint was created. If you specify the endpoint name using the **`-Name`** parameter, as in the preceding example, you will only get the information relevant to the specified endpoint.

We will have a deeper look into the possible session configuration and registering parameters in ***Chapter 10***, *Language Modes and Just Enough Administration (JEA)*.

PowerShell remoting authentication and security considerations

PSRemoting traffic is encrypted by default – regardless of whether a connection was initiated via HTTP or HTTPS. The underlying protocol that's used is WS-Man, which is decoupled to allow it to be used more broadly. PSRemoting uses an authentication protocol, such as Kerberos or NTLM, to authenticate the session traffic, and SSL/TLS is used to encrypt the session traffic, regardless of whether the connection was initiated via HTTP or HTTPS.

But similar to every other computer, PSRemoting is only as secure as the computer that's been configured. And if you don't secure your administrator's credentials, an attacker can extract and use them against you.

Therefore, you should also put effort into hardening your infrastructure and securing your most valuable identities. You will learn more about Active Directory security and credential hygiene in ***Chapter 6***, *Active Directory – Attacks and Mitigations*, and learn more about what mitigations you can put in place in *Part 3*, *Securing PowerShell – Effective Mitigations in Detail*.

It's important to understand that enabling PSRemoting does not automatically ensure a secure environment. As with any remote management technology, it's critical to harden your systems and take appropriate security measures to protect against potential threats. This applies not only to PSRemoting but also to other remote management technologies, such as RDP. By investing time and effort into securing your systems and environment, you can mitigate potential risks and better protect your organization's assets.

First, let's have a look at how authentication is used within PSRemoting.

Authentication

By default, WinRM uses **Kerberos** for authentication and falls back to **NTLM** in case Kerberos authentication is not possible.

When used within a domain, Kerberos is the standard to authenticate. To use Kerberos for authentication in PSRemoting, ensure that both the client and server computers are connected to the same domain and that the DNS names have been properly configured and are reachable. It's also important to note that from a Kerberos perspective, the server must be registered in Active Directory.

In general, you can specify which protocol should be used when connecting to a remote computer:

```
> Enter-PSSession -ComputerName PSSEC-PC01 -Authentication Kerberos
```

When establishing a PSRemoting session, if the **`-Authentication`** parameter is not specified, the default value of **`Default`** is used, which is equal to the **`Negotiate`** value. This means that the client and server negotiate the best authentication protocol to use based on what is supported by both systems.

Typically, *Kerberos* is the preferred protocol, but if it's not available or supported, the system will fall back to using *NTLM*. More information about **Negotiate** can be found in the Microsoft documentation for Negotiate in Win32 applications: **https://learn.microsoft.com/en-us/windows/win32/secauthn/microsoft-negotiate**.

What are the circumstances for an NTLM fallback?

PSRemoting was designed to work with Active Directory, so Kerberos is the preferred authentication protocol. But in some cases, Kerberos authentication is not possible and NTLM is used.

Kerberos:

- Computers are joined to the same domain or are both within domains that trust each other.
- The client can resolve the server's hostname or IP address.
- The server has a valid **Service Principal Name** (**SPN**) registered in Active Directory. The SPN matches the target you are connecting to.

NTLM:

- Commonly used to connect to non-domain-joined workstations
- If IP addresses are used instead of DNS names

To connect to the **`PSSec-PC01`** computer via Kerberos, we can use the following command:

```
> Enter-PSSession -ComputerName PSSec-PC01
```

If no credentials were explicitly specified, if the current user has permission to access the remote computer, and if the remote computer is configured to accept Kerberos authentication, the connection will be established automatically without the need to provide any explicit credentials. This is one of the benefits of using Kerberos authentication, as the authentication process is implicit and seamless for the user.

If the current user does not have permission to access the remote computer, we can also specify explicitly which credentials should be used with the **`-Credential`** parameter. To simplify testing, we use **`Get-Credential`** to prompt for the credentials and store them in the **`$cred`** secure string:

```
$cred = Get-Credential -Credential "PSSEC\Administrator"
```

Then, we connect via Kerberos:

```
Enter-PSSession -ComputerName PSSEC-PC01 -Credential $cred
```

If you capture the traffic using Wireshark, you will see that WinRM includes Kerberos as its **`content-type`** as part of its protocol, indicating that Kerberos was used for authentication. While the actual Kerberos traffic itself may not be visible in the HTTP packet, the use of Kerberos for authentication can still be confirmed by examining the headers in the WinRM traffic. Additionally, you can see that the entire HTTP session is encrypted, providing an added layer of security:

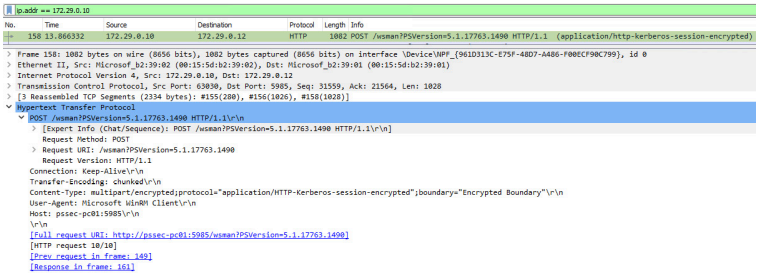

Figure 3.14 – WinRM HTTP traffic captured with Wireshark

As you can see, a session to **`PSSec-PC01`** has been established over port **`5985`** (WinRM over HTTP), using PowerShell version 5.1.17763.1490. The request was sent via WS-Man.

Once the initial authentication process is complete, WinRM proceeds to encrypt all ongoing communication to maintain the security of the data being exchanged between the client and server. When establishing a connection over HTTPS, the TLS protocol is utilized to negotiate the encryption method used for data transportation. In the case of an HTTP connection, the encryption that's utilized for message-level encryption is determined by the initial authentication protocol used.

The level of encryption provided by each authentication protocol is as follows:

- **Basic authentication**: No encryption.
- **NTLM authentication**: RC4 cipher with a 128-bit key.
- **Kerberos authentication**: **etype** in the TGS ticket determines the encryption. On modern systems, this is typically AES-256.
- **CredSSP authentication**: The TLS cipher suite that was negotiated in the handshake will be used.

Note that while the HTTP protocol is used as the connection protocol, the content is encrypted using the appropriate encryption mechanism based on the initial authentication protocol used. A common misconception about PSRemoting is that a connection using WinRM over HTTP is not encrypted. However, as you can see in the following screenshot, this is not the case:

Figure 3.15 – Kerberos TCP stream captured with Wireshark

If DNS names are not working and if both hosts are not joined to the same domain, NTLM will be used as a fallback option.

If you are connecting to a remote computer in the same domain, with working DNS names, NTLM is still used to connect if the host IP address is specified instead of the hostname:

```
Enter-PSSession -ComputerName 172.29.0.12 -Credential $cred
```

Capturing the traffic with Wireshark once more reveals that NTLM was used to authenticate and that the traffic is encrypted as well:

Figure 3.16 – NTLM traffic captured with Wireshark

Similar to connecting with Kerberos, you can see that a connection is established to the host, **172.29.0.12**, using WinRM over HTTP (port **5985**). But this time, NTLM is used instead of Kerberos to negotiate the session.

Using NTLM, you can even capture the hostname, the username, the domain name, and the challenge, which is used for authentication.

Going deeper into the TCP stream, it becomes evident that the communication is once again encrypted, even when NTLM is used, as shown in the following screenshot:

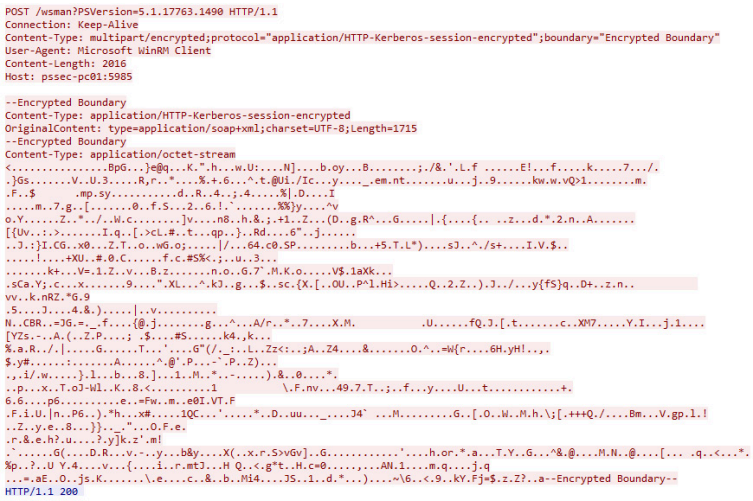

Figure 3.17 – NTLM TCP stream captured with Wireshark

When using NTLM authentication, please note that PSRemoting only works if the remote host was added to the **`TrustedHosts`** list.

When using NTLM authentication, it's important to understand the limitations of the **`TrustedHosts`** list. While adding a remote host to the **`TrustedHosts`** list can help you catch your mistakes, it's not a reliable way to ensure secure communication. This is because NTLM can't guarantee that you are connecting to the intended remote host, which makes using **`TrustedHosts`** misleading. It's important to note that the main weakness of NTLM is its inability to verify the identity of the remote host. Therefore, even with **`TrustedHosts`**, NTLM connections shouldn't be considered more trustworthy.

If the host is not specified as a trusted host and if the credentials are not explicitly provided (like we did when using **`-Credential $cred`**), establishing a remote session or running commands remotely will fail and show an error message:

```
> Enter-PSSession -ComputerName 172.29.0.10
Enter-PSSession : Connecting to remote server 172.29.0.10 failed with the following error message
cannot process the request. If the authentication scheme is different from Kerberos, or if the cli
joined to a domain, then HTTPS transport must be used or the destination machine must be added to
configuration setting. Use winrm.cmd to configure TrustedHosts. Note that computers in the Trusted
be authenticated. You can get more information about that by running the following command: winrm
more information, see the about_Remote_Troubleshooting Help topic.
At line:1 char:1
+ Enter-PSSession -ComputerName 172.29.0.10
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (172.29.0.10:String) [Enter-PSSession], PSRemotingT
    + FullyQualifiedErrorId : CreateRemoteRunspaceFailed
```

Kerberos and NTLM are not the only authentication protocols, but they are the most secure compared with others. Let's have a look at what other methods exist and how you can enforce them.

Authentication protocols

Of course, it is also possible to configure which authentication method should be used by specifying the `-Authentication` parameter.

AUTHENTICATION PROTOCOLS

If it is possible to use Kerberos authentication, you should always use Kerberos, as this protocol provides most security features.

*Proceed to **Chapter 6**, Active Directory – Attacks and Mitigation, to learn more about authentication and how Kerberos and NTLM work.*

The following are all accepted values for the `-Authentication` parameter:

- `Default`: This is the default value. Here, `Negotiate` will be used.
- `Basic`: Basic authentication is used to authenticate, using the HTTP protocol, but does not provide security by itself – neither for the data, which is transported in cleartext over the network, nor for the credentials. However, when paired with TLS, this can still be a reasonably secure mechanism and is commonly used by many websites.

As the credentials are only encoded using Base64 encoding, the encryption can easily be reversed and the credentials can be extracted in cleartext.

This authentication does not provide confidentially for the provided credentials if they're not encrypted with **SSL**/**TLS**.

- `Credssp`: Using the **CredSSP** authentication, the user's credentials will be provided by PowerShell from the client to the remote server to authenticate the user. This mode is particularly useful in situations where you need the remote session to be able to authenticate as you for further network hops. After this authentication, the credentials are passed between the client and server in an encrypted format to maintain security.

When using the CredSSP authentication mechanism, PowerShell passes the user's full credentials to the remote server for authentication. This means that if you connect to a compromised machine, an adversary can extract your credentials directly from memory. It's important to note that this is the default authentication mechanism of RDP, making PSRemoting a more secure alternative.

- `Digest`: Digest authentication is one of the methods a web server can use for authentication. The username and password are hashed using **MD5** cryptography algorithms before they're sent over the network using the **HTTP** protocol. Before hashing, a nonce is added to avoid replay attacks.

It does not provide strong authentication compared to other authentication protocols (for example, key-based ones), but it is still stronger than weaker authentication mechanisms and should be considered as a replacement for weak basic authentication.

- **`Kerberos`**: This form of authentication uses the Kerberos protocol. Kerberos is the standard to authenticate in a domain and provides the highest security.
- **`Negotiate`**: This option allows the client to negotiate the authentication. When a domain account is used, the authentication will be via Kerberos; with a local account, it falls back to NTLM.
- **`NegotiateWithImplicitCredential`**: This option uses the current user's credentials to authenticate (run as).

These authentication mechanisms can be used within all PSRemoting cmdlets.

They are also specified in the **`AuthenticationMechanism` enum**, which is defined in Microsoft docs: https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.runspaces.authenticationmechanism.

It's important to note that PowerShell considers some authentication mechanisms as potentially dangerous and may show error messages if you try to use them. In such cases, you would need to explicitly override these errors to proceed with the dangerous authentication mechanism.

Basic authentication security considerations

If used without any additional encryption layers, basic authentication is not secure. In this section we are going to explore a very good example of why you should not use basic authentication or why you should always encrypt your communication using **Transport Layer Security** (**TLS**) if you have to use basic authentication.

CAUTION!

Do not configure this in your production environment as this configuration is highly insecure and is only shown for testing purposes. You will compromise yourself if you use this configuration!

If you want to configure your **test environment** to use basic authentication and allow unencrypted traffic, you need to configure your WinRM configuration to allow basic authentication, as well as unencrypted traffic.

In this example, **`PSSec-PC01`** is the remote host to which we want to connect using unencrypted traffic and basic authentication. We will connect from a management machine, which will be **`PSSec-PC02`**.

When we try to authenticate from **`PSSec-PC02`** to **`PSSec-PC01`** (the IP address is **`172.29.0.12`**) using the **`-Authentication Basic`** parameter, we

get a message stating that we need to provide a username and a password to authenticate using basic authentication:

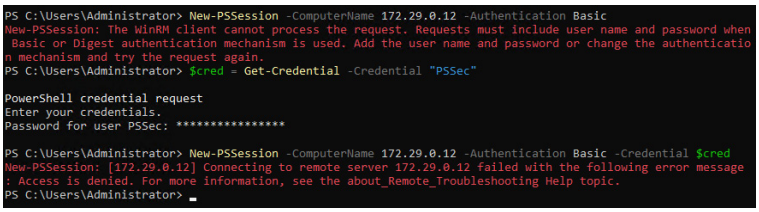

Figure 3.18 – Error messages are shown if an insecure authentication mechanism is used

Once we provide these credentials, we are still not able to authenticate and get another error message stating that access has been denied. The reason for this is that **basic** authentication is an insecure authentication mechanism if it's not protected by TLS. Therefore, PSRemoting does not allow you to connect using this insecure authentication mechanism if you don't configure it explicitly.

So, let's configure basic authentication explicitly in our demo setup, knowing that we will weaken our configuration on purpose. First, allow unencrypted traffic on **PSSec-PC01**:

```
> winrm set winrm/config/service '@{AllowUnencrypted="true"}'
```

Remember to differentiate between **service** and **client** configuration. As we want to connect to **PSSec-PC01**, we will connect to the WinRM service, so we are configuring **service**.

Next, configure basic authentication to be allowed:

```
> winrm set winrm/config/service/auth '@{Basic="true"}'
```

After making changes to the WinRM configuration, it is important to restart the WinRM service for the new configuration to take effect:

```
> Restart-Service -Name WinRM
```

Now, let's configure **PSSec-PC02** to establish unencrypted connections to other devices using basic authentication.

First, we must configure the client so that unencrypted connections can be initialized:

```
> winrm set winrm/config/client '@{AllowUnencrypted="true"}'
```

Then, we must make sure that the client is allowed to establish connections using basic authentication:

```
> winrm set winrm/config/client/auth '@{Basic="true"}'
```

Lastly, restart the WinRM service to load the new configuration:

```
> Restart-Service -Name WinRM
```

Again, this configuration exposes your devices and makes them vulnerable. Specifically, it exposes your credentials to potential attackers who could intercept network traffic while you connect to your machines. This could allow an attacker to gain unauthorized access to your systems and potentially compromise sensitive data or perform malicious actions.

Therefore, we apply this configuration only in a test environment. In productive environments, it's important to take appropriate security measures, such as enabling encryption and using secure authentication protocols, to protect your devices and data.

As soon as we have our vulnerable configuration in place, it's time to connect using basic authentication. I have added a local user called **PSSec** on **PSSec-PC01**, which I will use in this example.

Let's connect from **PSSec-PC02** to **PSSec-PC01** (the IP address is **172.29.0.12**) by using the **-Authentication** parameter while specifying **Basic**, as well as the credentials for the **PSSec** user:

```
> $cred = Get-Credential -Credential "PSSec"
> New-PSSession -ComputerName 172.29.0.12 -Authentication Basic -Credential $cred
```

The session is being established. If I track the traffic using Wireshark, I will see the SOAP requests that are being made. Even worse, I can see the **Authorization** header, which exposes the Base64-encrypted username and password:

```
POST /wsman?PSVersion=5.1.17763.1490 HTTP/1.1
Connection: Keep-Alive
Content-Type: application/soap+xml;charset=UTF-8
User-Agent: Microsoft WinRM Client
Content-Length: 1667
Host: 172.29.0.12:5985
Authorization: Basic UFNTZWM6UFMtU2VjUm9ja3oxMjM0IQ==

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:w="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns:p="http://schemas.microsoft.com/wbem/wsman/1/
wsman.xsd"><s:Header><a:To>http://172.29.0.12:5985/wsman?PSVersion=5.1.17763.1490</a:To><w:ResourceURI
s:mustUnderstand="true">http://schemas.microsoft.com/powershell/Microsoft.PowerShell</w:ResourceURI><a:ReplyTo><a:Address
s:mustUnderstand="true">http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</a:Address></a:ReplyTo><a:Action
s:mustUnderstand="true">http://schemas.microsoft.com/wbem/wsman/1/windows/shell/Receive</a:Action><w:MaxEnvelopeSize
s:mustUnderstand="true">512000</w:MaxEnvelopeSize><a:MessageID>uuid:364E7B0F-DBA3-4E81-A5E6-E702FDF0DDAE</
a:MessageID><w:Locale xml:lang="en-US" s:mustUnderstand="false" /><p:DataLocale xml:lang="de-DE" s:mustUnderstand="false" /
><p:SessionId s:mustUnderstand="false">uuid:65990572-516D-427E-A2AB-93614F276205</p:SessionId><p:OperationID
s:mustUnderstand="false">uuid:AA660375-7929-4BF7-B1D1-4D4E7AF7D946</p:OperationID><p:SequenceId
s:mustUnderstand="false">1</p:SequenceId><w:SelectorSet><w:Selector Name="ShellId">06483E88-EF48-43BF-815C-E3DD34CEADEB</
w:Selector></w:SelectorSet><w:OptionSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><w:Option
Name="WSMAN_CMDSHELL_OPTION_KEEPALIVE">TRUE</w:Option></w:OptionSet><w:OperationTimeout>PT180.000S</w:OperationTimeout></
s:Header><s:Body><rsp:Receive xmlns:rsp="http://schemas.microsoft.com/wbem/wsman/1/windows/shell"
SequenceId="0"><rsp:DesiredStream>stdout</rsp:DesiredStream></rsp:Receive></s:Body></s:Envelope>HTTP/1.1 200
Content-Type: application/soap+xml;charset=UTF-8
Server: Microsoft-HTTPAPI/2.0
Date: Sun, 10 Jan 2021 15:29:54 GMT
Content-Length: 2264

<s:Envelope xml:lang="en-US" xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://schemas.xmlsoap.org/ws/
2004/08/addressing" xmlns:w="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns:rsp="http://schemas.microsoft.com/wbem/
wsman/1/windows/shell" xmlns:p="http://schemas.microsoft.com/wbem/wsman/1/wsman.xsd"><s:Header><a:Action>http://
schemas.microsoft.com/wbem/wsman/1/windows/shell/ReceiveResponse</a:Action><a:MessageID>uuid:FFB2E649-B93F-4125-B4DE-
F0EE9C18C67E</a:MessageID><a:To>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</a:To><p:OperationID
s:mustUnderstand="false">uuid:AA660375-7929-4BF7-B1D1-4D4E7AF7D946</p:OperationID><p:SequenceId>1</
p:SequenceId><a:RelatesTo>uuid:364E7B0F-DBA3-4E81-A5E6-E702FDF0DDAE</a:RelatesTo></
s:Header><s:Body><rsp:ReceiveResponse><rsp:Stream
Name="stdout">3gDeAAAAAAAAAAABAAAAAAAAAADAAAAygEAAAACAAEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADvu788T2JqIFJlZklkPSIwI
j48TVM+PFZlcnNpb24gTj0icHJvdG9jb2x2ZXJzaW9uIj4yLjM8L1ZlcnNpb24+PFZlcnNpb24gTj0iUFNWZXJzaW9uIj4yLjA8L1ZlcnNpb24+PFZlcnNpb24g
```

Figure 3.19 – Wireshark capture of authenticating using unencrypted basic authentication

Base64 can be easily decrypted, for example, with PowerShell itself:

```
PS C:\Users\Administrator> [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String
("UFNTZWM6UFMtU2VjUm9ja3oxMjM0IQ=="))
PSSec:PS-SecRockz1234!
```

Figure 3.20 – Decrypting Base64-encrypted credentials

So, an attacker can easily find out that the password of the **PSSec** user is **PS-SecRockz1234!** and can either inject the session as a man in the middle or use the password to impersonate the **PSSec** user – a great start when they're attacking the entire environment.

I hope I made the risks of basic authentication and unencrypted sessions more transparent so that you will try this configuration in test environments only – and avoid it in production.

PowerShell remoting and credential theft

Depending on the authentication method that is used, credentials can be entered into the remote system, which can be stolen by an adversary. If you are interested in learning more about **credential theft** and mitigations, the *Mitigating Pass-the-Hash (PtH) Attacks and Other Credential Theft* white papers are a valuable resource: **https://www.microsoft.com/en-us/download/details.aspx?id=36036**.

By default, PSRemoting does not leave credentials on the target system, which makes PowerShell an awesome administration tool.

But if, for example, PSRemoting with CredSSP is used, the credentials enter the remote system, where they can be extracted and used to impersonate identities.

Keep in mind that when using CredSSP as an authentication mechanism, the credentials used to authenticate to the remote system are cached on that system. While this is convenient for single sign-on purposes, it also makes those cached credentials vulnerable to theft. If you can avoid it, do not use CredSSP as an authentication mechanism. But if you choose to use CredSSP, it is recommended that you enable Credential Guard to help mitigate this risk.

We will have a closer look at authentication and how the infamous pass-the-hash attack works in ***Chapter 6***, *Active Directory – Attacks and Mitigation*.

Executing commands using PowerShell remoting

Sometimes, you may want to run a command remotely but have not configured PSRemoting. Some cmdlets provide built-in remoting technologies that can be leveraged.

All commands that offer a built-in remoting technology have one thing in common: typically, they all have a parameter called **`-ComputerName`** to specify the remote endpoint.

To get a list of locally available commands that have the option to run tasks remotely, use the **`Get-Command -CommandType Cmdlet -ParameterName ComputerName`** command:

```
> Get-Command -ParameterName ComputerName
CommandType  Name              Version    Source
-----------  ----              -------    ------
Cmdlet       Connect-PSSession 3.0.0.0    Microsoft.PowerShell.Core
Cmdlet       Enter-PSSession   3.0.0.0    Microsoft.PowerShell.Core
```

```
Cmdlet       Get-PSSession      3.0.0.0    Microsoft.PowerShell.Core
Cmdlet       Invoke-Command     3.0.0.0    Microsoft.PowerShell.Core
Cmdlet       New-PSSession      3.0.0.0    Microsoft.PowerShell.Core
Cmdlet       Receive-Job        3.0.0.0    Microsoft.PowerShell.Core
Cmdlet       Receive-PSSession  3.0.0.0    Microsoft.PowerShell.Core
Cmdlet       Remove-PSSession   3.0.0.0    Microsoft.PowerShell.Core
```

Please note that this list is not complete.

Cmdlets with a **`-ComputerName`** parameter do not necessarily use WinRM. Some use WMI, many others use RPC – it depends on the underlying technology of the cmdlet.

As every cmdlet has an underlying protocol, its firewall configuration and services need to be configured accordingly. This could mean a big management overhead. So, when managing environments remotely, it makes sense to configure PSRemoting accordingly: using WinRM is firewall-friendly and easier to configure and maintain.

DO NOT BE CONFUSED!

PSRemoting should not be confused with using the **`-ComputerName`** *parameter of a cmdlet to execute it on a remote computer. They are distinct approaches with different capabilities and usage scenarios. Those cmdlets that utilize the* **`-ComputerName`** *parameter rely on their underlying protocols, which often need a separate firewall exception rule to run.*

Executing single commands and script blocks

You can *execute a single command* or *entire script blocks* on a remote or local computer using the **`Invoke-Command`** cmdlet:

```
Invoke-Command -ComputerName <Name> -ScriptBlock {<ScriptBlock>}
```

The following example shows how to restart the printer spooler on the **`PSSec-PC01`** remote computer, which is displaying verbose output:

```
> Invoke-Command -ComputerName PSSec-PC01 -ScriptBlock { Restart-Service -Name Spooler -Verbose }
VERBOSE: Performing the operation "Restart-Service" on target "Print Spooler (Spooler)".
```

`Invoke-Command` is a great option for running local scripts and commands on a remote computer.

If you don't want to copy the same scripts to your remote machine(s), you can use **`Invoke-Command`** with the **`-FilePath`** parameter to *run the local script on the remote system*:

```
> Invoke-Command -ComputerName PSSec-PC01 -FilePath c:\tmp\test.ps1
```

When using the **`-FilePath`** parameter with **`Invoke-Command`**, it is important to keep in mind that any dependencies required by the script (such as other scripts or commands) must also be present on the remote system. Otherwise, the script will not run as expected.

You can also *execute commands on multiple systems* – just specify all the remote systems that you want to execute your command or script on in the **`-ComputerName`** parameter. The following command restarts the print spooler on **`PSSec-PC01`** and **`PSSec-PC02`**:

```
> Invoke-Command -ComputerName PSSec-PC01,PSSec-PC02 {Restart-Service -Name Spooler}
```

Please have a look at the official PowerShell documentation to learn all options that **`Invoke-Command`** has to offer: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/invoke-command**.

Working with PowerShell sessions

The **`-Session`** parameter indicates that a cmdlet or function supports sessions within PSRemoting.

To find all locally available commands that support the **`-Session`** parameter, you can use the **`Get-Command -ParameterName session`** command:

```
PS C:\Users\Administrator> Get-Command -ParameterName session

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Connect-PSSession                                  7.2.9.500  Microsoft.PowerShell…
Cmdlet          Disconnect-PSSession                               7.2.9.500  Microsoft.PowerShell…
Cmdlet          Enter-PSSession                                    7.2.9.500  Microsoft.PowerShell…
Cmdlet          Invoke-Command                                     7.2.9.500  Microsoft.PowerShell…
Cmdlet          New-PSSession                                      7.2.9.500  Microsoft.PowerShell…
Cmdlet          Receive-Job                                        7.2.9.500  Microsoft.PowerShell…
Cmdlet          Receive-PSSession                                  7.2.9.500  Microsoft.PowerShell…
Cmdlet          Remove-PSSession                                   7.2.9.500  Microsoft.PowerShell…
```

Figure 3.21 – All commands that provide a session parameter

All local commands that provide a **`-Session`** parameter will be shown.

Interactive sessions

By leveraging the **`Enter-PSSession`** command, you can initiate an interactive session. Once the session has been established, you can work on the remote system's shell:

```
Administrator: PowerShell 7 (x64)
PS C:\Users\Administrator> $cred = Get-Credential -Credential "PSSEC\Administrator"

PowerShell credential request
Enter your credentials.
Password for user PSSEC\Administrator: **************

PS C:\Users\Administrator> Enter-PSSession -ComputerName PSSec-PC01 -Credential $cred
[PSSec-PC01]: PS C:\Users\Administrator\Documents> hostname
PSSec-PC01
[PSSec-PC01]: PS C:\Users\Administrator\Documents> Restart-Service -Name Spooler -Verbose
VERBOSE: Performing the operation "Restart-Service" on target "Print Spooler (Spooler)".
WARNING: Waiting for service 'Print Spooler (Spooler)' to start...
[PSSec-PC01]: PS C:\Users\Administrator\Documents> Exit-PSSession
PS C:\Users\Administrator>
```

Figure 3.22 – Entering a PowerShell session, executing a command, and exiting the session

Once your work is finished, use **`Exit-PSSession`** to close the session and the remote connection.

Persistent sessions

The **`New-PSSession`** cmdlet can be utilized to establish a persistent session.

As in a former example, we use **`Get-Credential`** once more to store our credentials as a secure string in the **`$cred`** variable.

Using the following command, we create two sessions for the **PSSec-PC01** and **PSSec-PC01** remote computers to execute commands:

```
$sessions = New-PSSession -ComputerName PSSec-PC01, PSSec-PC02 -Credential $cred
```

To display all active sessions, you can use the **Get-PSSession** command:

```
PS C:\Windows\System32> $cred = Get-Credential -Credential "PSSEC\Administrator"

PowerShell credential request
Enter your credentials.
Password for user PSSEC\Administrator: **************

PS C:\Windows\System32> $sessions = New-PSSession -ComputerName PSSec-PC01, PSSec-PC02 -Credential
 $cred
PS C:\Windows\System32> Get-PSSession

 Id Name            Transport ComputerName    ComputerType    State         ConfigurationName
 -- ----            --------- ------------    ------------    -----         -----------------
  1 Runspace1       WSMan     PSSec-PC01      RemoteMachine   Opened        Microsoft.PowerShell
  2 Runspace2       WSMan     PSSec-PC02      RemoteMachine   Opened        Microsoft.PowerShell
```

Figure 3.23 – Creating persistent sessions and displaying them

Now, you can use the **$sessions** variable to run commands in all remote computer sessions that you've specified.

A common use case is to check whether all security updates were applied to your remote computers. In this case, we want to check whether the **KB5023773** hotfix is installed on all remote computers. We also don't want any error messages to be displayed if the hotfix was not found, so we will use the **-ErrorAction SilentlyContinue** parameter in our code snippet:

```
Invoke-Command –Session $sessions -ScriptBlock { Get-Hotfix -Id 'KB5023773' -ErrorAction SilentlyC
```

The following is the output we get after running this command:

```
PS C:\Windows\System32> Invoke-Command -Session $sessions -ScriptBlock { Get-Hotfix -Id 'KB5023773' -ErrorAction
SilentlyContinue }

Source        Description      HotFixID      InstalledBy          InstalledOn              PSComputerName
------        -----------      --------      -----------          -----------              --------------
PSSEC-PC01    Update           KB5023773     NT AUTHORITY\SYSTEM  06.04.2023 00:00:00      PSSec-PC01

PS C:\Windows\System32>
```

Figure 3.24 – Running a command in all specified sessions

As it turns out, the hotfix is only installed on **PSSec-PC01** but is missing on the second computer, **PSSec-02**.

To act on this and install the missing update, we can either send more commands directly into the session or we can enter the session interactively by specifying the session ID – that is, **Enter-PSSession -Id 2**:

```
PS C:\Windows\System32> Enter-PSSession -Id 2
[PSSec-PC02]: PS C:\Users\Administrator\Documents> Get-WindowsUpdate -Install -KBArticleID 'KB5023773'
[PSSec-PC02]: PS C:\Users\Administrator\Documents> Exit-PSSession
PS C:\Windows\System32>
```

Figure 3.25 – Entering a persistent session, running a command, and exiting it again

Now that we have entered the session, we can run the **Get-WindowsUpdate** command to install the missing update. Please note that this command is not available by default and requires you to install the **PSWindowsUpdate** module:

```
Get-WindowsUpdate -Install -KBArticleID 'KB5023773'
```

After our command has run, we can exit the session using **`Exit-PSSession`**, which only disconnects us from the session but leaves the session open.

NOTE

If you are using an interactive session, all executed modules, such as **`PSWindowsUpdate`**, *need to be installed on the remote system. If you use* **`Invoke-Command`** *to run commands in a persistent session, the module only needs to be installed on the computer that you use to run the commands:*

Invoke-Command – Session $sessions -ScriptBlock { Get-WindowsUpdate -Install -KBArticleID 'KB5023773'}

If we check for **`KB5023773`** after some time, we will see that the update was installed:

```
PS C:\Windows\System32> Invoke-Command -Session $sessions -ScriptBlock { Get-Hotfix -Id 'KB5023773' -ErrorAction
 SilentlyContinue }

Source        Description      HotFixID      InstalledBy          InstalledOn              PSComputerName
------        -----------      --------      -----------          -----------              --------------
PSSEC-PC01    Update           KB5023773     NT AUTHORITY\SYSTEM  06.04.2023 00:00:00      PSSec-PC01
PSSEC-PC02    Update           KB5023773     NT AUTHORITY\SYSTEM  06.04.2023 00:00:00      PSSec-PC02

PS C:\Windows\System32>
```

Figure 3.26 – The update was installed successfully

As soon as we are finished with our work and if we don't need our sessions anymore, we can remove them using the **`Remove-PSSession`** command:

- Here, we can use the **`$sessions`** variable, which we specified earlier:

  ```
  Remove-PSSession -Session $sessions
  ```

- Alternatively, we can remove a single session by using the **`-id`** parameter:

  ```
  Remove-PSSession -id 2
  ```

After removing one or all session(s), you can use **`Get-PSSession`** to verify this:

```
PS C:\Windows\System32> Remove-PSSession -Session $sessions
PS C:\Windows\System32> Get-PSSession
PS C:\Windows\System32>
```

Figure 3.27 – Removing all persistent sessions

Executing commands using PSRemoting can simplify your daily administration workload immensely. Now that you have learned the basics, you can combine it with your PowerShell scripting knowledge. What problems will you solve and what tasks will you automate?

Best practices

To ensure optimal security and performance when using PSRemoting, it's important to follow the best practices enforced by the product. These practices are designed to minimize the risk of security breaches and ensure that your remote management tasks run smoothly.

Authentication:

- If possible, use only Kerberos or NTLM authentication.
- Avoid CredSSP and basic authentication whenever possible.
- In the best case, restrict the usage of all other authentication mechanisms besides Kerberos/NTLM.
- SSH remoting – configure public key authentication and keep the private key protected.

Limit connections:

- Limit connections via firewall from a management subnet (hardware and software if possible/available).

PSRemoting's default firewall policies differ based on the network profile. In a **Domain**, **Workgroup**, or **Private** network profile, PSRemoting is available to all by default (assuming they have valid credentials). In a **Public** profile, PSRemoting refuses to listen to that adapter by default. If you force it to, the network rule will limit access to only systems on the same network subnet.

- Use a secure management system to manage systems via PSRemoting. Consider limiting connections from a management **virtual network (VNet)** if you have one, which also applies to other management protocols such as RDP, WMI, CIM, and others.
- Use a secure management system to manage systems via PSRemoting. Use the clean source principle to build the management system and use the recommended privileged access model:
  - **https://learn.microsoft.com/en-us/security/privileged-access-workstations/privileged-access-success-criteria#clean-source-principle**
  - **https://learn.microsoft.com/en-us/security/privileged-access-workstations/privileged-access-access-model**

Restrict sessions:

- Use constrained language and JEA.
- You will learn more about JEA, constrained language, session security, and SDDLs in ***Chapter 10***, *Language Modes and Just Enough Administration (JEA).*

Audit insecure settings:

- Use the WinRM group policy to enforce secure PSRemoting settings on all managed systems, including encryption and authentication requirements.
- **`Get-Item WSMan:\localhost\Client\AllowUnencrypted`**: This setting should *not* be set to **`$true`**.
- Audit insecure WinRM settings regularly to ensure compliance with security policies:

```
Get-Item WSMan:\localhost\client\AllowUnencrypted
Get-Item wsman:\localhost\service\AllowUnencrypted
```

```
Get-Item wsman:\localhost\client\auth\Basic
Get-Item wsman:\localhost\service\auth\Basic
```

- Eventually, use **Desired State Configuration** (**DSC**) to audit and apply your settings.

And all other mitigation methods mentioned in the previous chapter, especially the following:

- Enable logging and transcription and monitor event logs. You can read more about this in ***Chapter 4***, *Detection – Auditing and Monitoring*.
- Eliminate unnecessary local and domain administrators
- Enable and enforce script signing. You will learn more about script signing in ***Chapter 11***, *AppLocker, Application Control, and Code Signing*.
- Configure **DSC** to harden your systems and control your system configuration.

PSRemoting is a great way to administrate your systems efficiently. Of course, it is only as secure as you configure it to be. If the right configuration is in place, administration via PSRemoting is even more secure than logging in interactively.

Summary

After reading this chapter, you should be familiar with how to use PowerShell remotely, using PSRemoting. You learned what options exist in PowerShell to establish remote connections, which enables you to not only manage Windows machines but also other operating systems, such as macOS and Linux.

You also learned what endpoints are and can create basic custom endpoints. You will strengthen this ability later in ***Chapter 10***, *Language Modes and Just Enough Administration (JEA)*, but you already know the basics.

Then, you learned a lot about authentication protocols that can be used and even more about security considerations when working with those protocols. You should also be aware of how easily an adversary can obtain decrypted credentials if a weak authentication protocol is used.

You should now be able to configure PSRemoting manually and centrally, which helps you set up your initial PSRemoting configuration in your production environment.

Last but not least, you learned how to execute commands using PSRemoting, which enables you to not only run one command on one device – you can also automate your tedious administration tasks.

When working with PowerShell – either remotely or locally – auditing and monitoring are very important topics. Using transcriptions and event logging helps the Blue Team detect adversaries and protect their environment.

Therefore, now that you are familiar with PSRemoting, we'll look at detection and logging within PowerShell in the next chapter.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, take a look at these resources.

Authentication:

- RFC 2617 – HTTP authentication (basic and digest authentication): **https://tools.ietf.org/html/rfc2617**
- **Credential Security Support Provider** (**CredSSP**) protocol:
  - **https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-cssp/85f57821-40bb-46aa-bfcb-ba9590b8fc30**
  - **https://ldapwiki.com/wiki/Wiki.jsp?page=CredSSP**
- Public key authentication:
  - **https://en.wikipedia.org/wiki/Public-key_cryptography**
  - **https://www.ssh.com/ssh/public-key-authentication**

CIM:

- CIM cmdlets: **https://devblogs.microsoft.com/powershell/introduction-to-cim-cmdlets/**
- CIM standard by DMTF: **https://www.dmtf.org/standards/cim**

DCOM:

- **DCOM** remote protocol: **https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dcom/4a893f3d-bd29-48cd-9f43-d9777a4415b0**

OMI:

- Open Management Infrastructure (OMI): **https://cloudblogs.microsoft.com/windowsserver/2012/06/28/open-management-infrastructure/**

Other useful resources:

- New-NetFirewallRule: **https://learn.microsoft.com/en-us/powershell/module/netsecurity/new-netfirewallrule**

PowerShell remoting:

- [MS-PSRP]: PowerShell remoting protocol: **https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-psrp/602ee78e-9a19-45ad-90fa-bb132b7cecec**
- Running Remote Commands: **https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/running-remote-**

commands

- WS-Man Remoting in PowerShell Core: **https://learn.microsoft.com/en-us/powershell/scripting/learn/remoting/wsman-remoting-in-powershell-core?view=powershell-7.3**
- WS-Man specifications by DMTF: **https://www.dmtf.org/standards/ws-man**
- WinRM security: **https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/winrmsecurity**
- PowerShell endpoints: **https://devblogs.microsoft.com/scripting/introduction-to-powershell-endpoints/**
- PSRemoting over SSH: https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/ssh-remoting-in-powershell-core
- The second hop: **https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/ps-remoting-second-hop**

WMI:

- Get-WmiObject: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-wmiobject**
- Invoke-WmiMethod: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/invoke-wmimethod**
- Register-WmiEvent: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/register-wmievent**
- Remove-WmiObject: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/remove-wmiobject**
- Set-WmiInstance: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/set-wmiinstance**

WS-Man:

- WS-Man standard by DMTF: **https://www.dmtf.org/standards/ws-man**
- WS-Management Remoting in PowerShell Core: https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/wsman-remoting-in-powershell-core

You can also find all the links mentioned in this chapter in the GitHub repository for ***Chapter 3*** – there's no need to manually type in every link: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter03/Links.md**.