

10

Language Modes and Just Enough Administration (JEA)

We have learned that PowerShell offers amazing logging and auditing capabilities and explored how to access the local system as well as Active Directory and Azure Active Directory. We also looked at daily red and blue team practitioner tasks. In this part of the book, we are diving deeper into mitigation features and how PowerShell can help you to build a robust and more secure environment.

We will first explore language modes and understand the difference between the Constrained Language mode and **Just Enough Administration (JEA)**. Then, we will dive deep into JEA and explore what is needed to configure your first very own JEA endpoint.

You will learn about the role capability and the session configuration file and learn how to deploy JEA in your environment. If you have the right tools at hand such as JEAAnalyzer, creating an initial JEA configuration is not too hard.

Finally, you will understand how to best leverage logging when working with JEA and which risky commands or bypasses you should avoid to harden your JEA configuration and your environment.

In this chapter, you will get a deeper understanding of the following topics:

- What are language modes within PowerShell?
- Understanding JEA
- Simplifying your deployment using JEAAnalyzer
- Logging within JEA sessions
- Best practices—avoiding risks and possible bypasses

Technical requirements

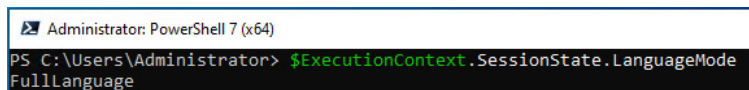
To get the most out of this chapter, ensure that you have the following:

- PowerShell 7.3 and above
- Visual Studio Code installed
- Access to the GitHub repository for **Chapter10**:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter10>

What are language modes within PowerShell?

A language mode in PowerShell determines which elements of PowerShell are allowed and can be used in a session. You can find out the language mode of the current session by running `$ExecutionContext.SessionState.LanguageMode`—of course, this only works if you are allowed to run this command:



```
Administrator: PowerShell 7 (x64)
PS C:\Users\Administrator> $ExecutionContext.SessionState.LanguageMode
FullLanguage
```

Figure 10.1 – Querying the language mode

In the example shown in the screenshot, the Full Language mode is enabled in the current session.

There are four different language modes available, which we will explore a little bit deeper in the following sections.

Full Language (FullLanguage)

The Full Language mode is the default mode for PowerShell. Every command and all elements are allowed.

The only restrictions that a user may experience would be if they do not have the Windows privileges to run a command (such as administrative privileges), but this behavior is not restricted by language mode.

Restricted Language (RestrictedLanguage)

The Restricted Language mode is a *data-specific form of the PowerShell language* that is primarily intended to support the localization files used by **Import-LocalizedData**. While cmdlets and functions can be executed in this mode, users are not allowed to run script blocks. It is important to note that the Restricted Language mode is not intended to be used explicitly in most scenarios and should only be used when working with localization files.

And beginning with PowerShell 7.2, the **New-Object** cmdlet is disabled if the system lockdown mode is configured.

Only the following variables are allowed by default:

- `$True`
- `$False`
- `$Null`
- `$PSCulture`
- `$PSUICulture`

Only the following operators are allowed by default:

- -eq
- -gt
- -lt

Please refer to [Chapter 2](#), *PowerShell Scripting Fundamentals*, for more details on operators.

No Language (NoLanguage)

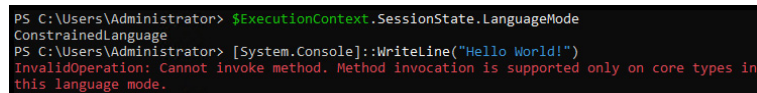
The No Language mode can be used via the API only and allows no single kind of script.

Similar to the Restricted Language mode, beginning with PowerShell 7.2, the **New-Object** cmdlet is disabled if the system lockdown mode is configured.

Constrained Language (ConstrainedLanguage)

As we learned earlier in the book in [Chapter 5](#), *PowerShell Is Powerful – System and API Access*, some of the most dangerous ways to abuse PowerShell are when COM or .NET are abused or if **Add-Type** is used to run and reuse code that was written in other languages (such as C#).

The Constrained Language mode prevents those dangerous scenarios, while it still permits the user to use legitimate .NET classes, as well as all cmdlets and PowerShell elements. It is designed to support day-to-day administrative tasks, but restricts the user from executing risky elements such as—for example—calling arbitrary APIs:



```
PS C:\Users\Administrator> $ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
PS C:\Users\Administrator> [System.Console]::WriteLine("Hello World!")
InvalidOperation: Cannot invoke method. Method invocation is supported only on core types in
this language mode.
```

Figure 10.2 – Running functions from arbitrary APIs is not possible within the constrained language mode

To configure a language mode for testing, you can simply set it via the command line:

```
> $ExecutionContext.SessionState.LanguageMode = "ConstrainedLanguage"
```

Using this particular setting in a production environment is not recommended—if an adversary gains access to the system, they could easily change this setting to compromise the security of the system:

```
> $ExecutionContext.SessionState.LanguageMode = "FullLanguage".
```

There is also the undocumented **__PSLockDownPolicy** environment variable that some blog posts recommend. However, this variable was only implemented for debugging and unit testing and should not be used for enforcement, due to the same reasons: an attacker can easily overwrite it, and it should only be used for testing.

To effectively use the Constrained Language mode to secure your PowerShell environment, it is critical to use it in conjunction with a robust application control solution such as **Windows Defender**

Application Control (WDAC):

<https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/select-types-of-rules-to-create>

Without such measures in place, attackers can easily bypass the Constrained Language mode by using other scripting engines or by creating custom malware in the form of **.exe** or **.dll** files.

We will also explore AppLocker and application control further in

Chapter 11, AppLocker, Application Control, and Code Signing.

Make sure to also refer to the PowerShell team's blog post on Constrained Language mode:

<https://devblogs.microsoft.com/powershell/powershell-constrained-language-mode/>

Constrained Language mode is a great option, but wouldn't it be great to also restrict which exact commands and parameters are allowed in a session or by a particular user? This is where JEA comes into play.

Understanding JEA

JEA does exactly what its name stands for: it allows you to define which role can execute which command and allows just enough administration rights.

Imagine you have multiple people working on one server system: there might be administrators and supporters who might need to perform certain operations such as restarting a service from time to time (for example, restarting the print spooler service on a print server). This operation would require administrative rights, but for the support person, an admin account would mean too many privileges—privileges that could be abused by an attacker in case the support person's credentials get stolen.

Using JEA, the system's administrator can define which commands can be run by a certain role and even restrict the parameters. As such, the support person can log in via **PowerShell Remoting (PSRemoting)**, quickly restart the print spooler service, and return to their daily business. No other commands can be used but those configured.

Additionally, JEA relies on PSRemoting, which is also a great way to avoid leaving credentials on the target system. There is even a possibility to configure that a virtual account is used on the target system on behalf of the operating person. Once the session is terminated, the virtual account will be destroyed and can no longer be used.

An overview of JEA

JEA relies on PSRemoting: a technology that lets you connect to defined endpoints remotely, which we explored further in [Chapter 3, Exploring PowerShell Remote Management Technologies and PowerShell Remoting](#).

There are two important files that you need to configure JEA basics—the **role capability file** and the **session configuration file**. Using these two files within a PSRemoting session allows JEA to let the magic work.

Of course, you also need to restrict all other forms of access (such as via Remote Desktop) to the target server to restrict users from bypassing your JEA restrictions.

The following diagram shows an overview of how a JEA connection works:

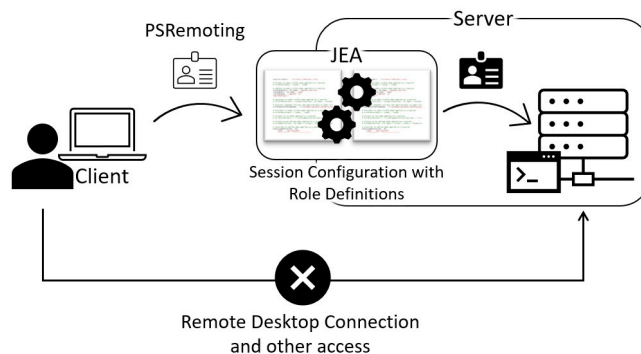


Figure 10.3 – High-level overview of how to connect with JEA

Using JEA even allows a non-administrative user to access a server to perform administrative tasks that were predefined for this user's role.

Depending on the configuration, a virtual account can be used on behalf of the user to allow non-administrative remote users to accomplish tasks that require administrative privileges. And don't worry; of course, every command that is executed under the virtual account is logged and can be mapped back to the originating user.

You might have heard much about PSRemoting sessions, but where in this picture can you find JEA?

Everything begins with starting an interactive session with a remote server—for example, by using **Enter-PSSession**.

There's also a possibility to add session options to the session—is this where you can find JEA? No, but session options come in very handy in case you don't want to connect to a *normal* PowerShell session. If you have, for example, a proxy to connect against, **-SessionOption** helps you to identify these details.

Session options are great, but they are not part of this chapter. So, if you want to learn more about them, refer to the options the **New-PSSessionOption** cmdlet provides:

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new->

psessionoption

Then, there is the option to add a configuration to the session, using the **-ConfigurationName** parameter. Is this where JEA hides? Well, almost, but we are not there yet. You can see in the following diagram the differences between options, configurations, and where JEA finally fits in:

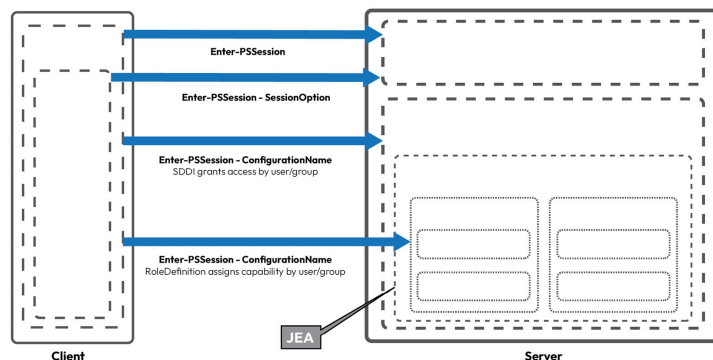


Figure 10.4 – Where JEA resides

JEA really comes into play within a configuration, where a role definition was created. So, JEA is a part of a session that is established, secured by **Security Descriptor Definition Language (SDDL)**, and with a special role definition. SDDLs define the rights a user or group can have to access a resource.

Planning for JEA

Before you can use JEA, there are a few things to consider first. JEA was included in PowerShell 5.0, so make sure that the right version is installed (5.0 or higher). You can check the current version using **\$PSVersionTable.PSVersion**.

Since JEA relies on PSRemoting and WinRM, make sure both are configured and enabled. See [Chapter 3, Exploring PowerShell Remote Management Technologies and PowerShell Remoting](#), for more details.

You also need administrative privileges on the system to be able to configure JEA.

And not only the right PowerShell version needs to be installed, but also the right operating system version. The following screenshot shows you all the supported versions for server operating systems, and what steps you need to take to make sure JEA is working properly:

Server Operating System	JEA Availability
Windows Server 2016 and above	Preinstalled
Windows Server 2012 R2	Full functionality with WMF 5.1
Windows Server 2012	Full functionality with WMF 5.1
Windows Server 2008 R2	Reduced functionality with WMF 5.1 JEA cannot be configured to use group managed service accounts. Virtual accounts and other JEA features are supported.

Figure 10.5 – JEA supportability for server operating systems

JEA can also be used on client operating systems. The following screenshot shows you which features are available with which version and what you need to do to get JEA running on each operating system:

Client Operating System	JEA Availability
Windows 10 1607 and above	Preinstalled
Windows 10 1603, 1511	Preinstalled, with reduced functionality Unsupported features: running as a group managed service account, conditional access rules in session configurations, the user drive, and granting access to local user accounts.
Windows 10 1507	Not available
Windows 8, 8.1	Full functionality with WMF 5.1
Windows 7	Reduced functionality with WMF 5.1 JEA cannot be configured to use group managed service accounts. Virtual accounts and other JEA features <i>are</i> supported.

Figure 10.6 – JEA supportability for client operating systems

Finally, you need to identify which users and/or groups you want to restrict and what rights each one of them should have. This might sound quite challenging. In this chapter, you will find some helpful tricks and tools to help you with this task.

But before we dive into that, let's explore what JEA consists of. First, there are two main files behind JEA, as follows:

- The role capability file
- The session configuration file

Let's first explore what the role capability file is about and how to configure it.

Role capability file

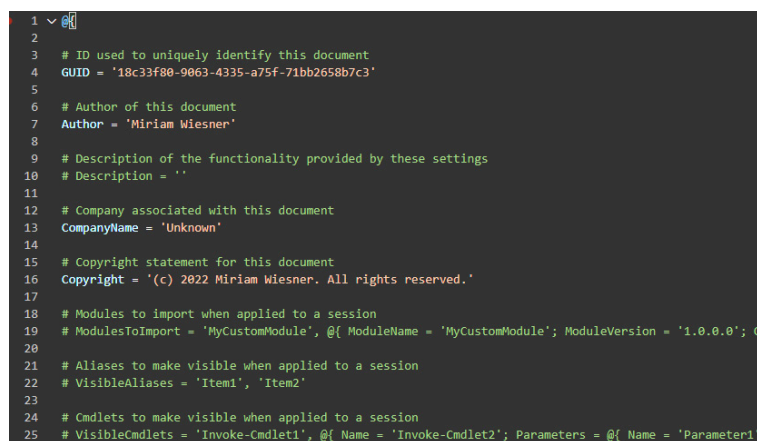
The role capability file determines which commands each role is allowed to run. You can specify which actions users in a particular role can perform and restrict these roles to using certain cmdlets, functions, providers, and external programs only.

It is common to define role capability files for certain roles—such as for print server admins, DNS admins, tier 1 helpdesk, and many more. Since role capability files can be implemented as part of PowerShell modules, you can easily share them with others.

Using **New-PSRoleCapabilityFile**, you can create your first skeleton JEA role capability file:

```
> New-PSRoleCapabilityFile -Path .\Support.psrc
```

An empty file, named **Support.psrc**, is created with prepopulated parameters that can be filled and edited:



```
1 1
2
3 # ID used to uniquely identify this document
4 GUID = '18c33f80-9063-4335-a75f-71bb2658b7c3'
5
6 # Author of this document
7 Author = 'Miriam Wiesner'
8
9 # Description of the functionality provided by these settings
10 # Description = ''
11
12 # Company associated with this document
13 CompanyName = 'Unknown'
14
15 # Copyright statement for this document
16 Copyright = '(c) 2022 Miriam Wiesner. All rights reserved.'
17
18 # Modules to import when applied to a session
19 # ModulesToImport = 'MyCustomModule', @( ModuleName = 'MyCustomModule'; ModuleVersion = '1.0.0.0'; G
20
21 # Aliases to make visible when applied to a session
22 # VisibleAliases = 'Item1', 'Item2'
23
24 # Cmdlets to make visible when applied to a session
25 # VisibleCmdlets = 'Invoke-Cmdlet1', @( Name = 'Invoke-Cmdlet2'; Parameters = @( Name = 'Parameter1'
```

Figure 10.7 – An empty skeleton role capability file

When choosing the name of the role capability file, make sure that it reflects the name of the actual role—so, be careful what name you choose for each file. In our example, we created the **Support.psrc** role capability file, which is a great start to configuring a support role.

You can find a generated skeleton file without any configuration in the GitHub repository of this book:

https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter10/JEA_SkeletonRoleCapabilityFile.psrc

Allowing PowerShell cmdlets and functions

Let's get started with an easy example of a role capability file. Let's imagine you are the administrator of an organization and the helpdesk reports regular issues with the print server. Well, one solution would be to give helpdesk administration privileges on all print servers, but that would give all helpdesk employees too many privileges on the print servers and probably expose your environment to risk.

Therefore, you might want to give the helpdesk employees only the privilege to restart services on the print servers.

You might have heard about **Restart-Service**, which serves exactly this purpose: to restart services. But was it a *cmdlet* or a *function*? Or even an *alias*?

If you are unsure about a certain command, the **Get-Command** cmdlet can help you in finding out more information:

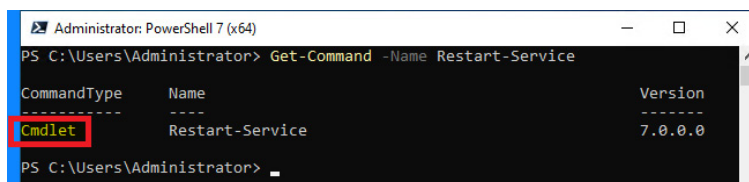


Figure 10.8 – Using Get-Command to find out the command type

Thanks to **Get-Command**, we now know that **Restart-Service** is a *cmdlet* and we can continue to configure it. If you have a look at the generated skeleton **.psrc** file, you can see multiple sections that start with **Visible**. Using these, you are able to define what will be available in your JEA session. All the parameters that you can configure in the role capability file align with the parameters that the **New-PSRoleCapabilityFile** cmdlet provides:

```
> New-PSRoleCapabilityFile -Path <path> -ParameterName <values>
```

For example, if you wanted to configure a simple JEA configuration and only make the **Restart-Service** cmdlet available, you could use the following command:

```
> New-PSRoleCapabilityFile -Path .\Support.psrc -VisibleCmdlets 'Restart-Service'
```

In this example, we used the **-VisibleCmdlets** parameter to configure the **Restart-Service** cmdlet to be available in the **Support** role, so let's have a closer look at what we can do using this configuration option.

VisibleCmdlets

Use the **-VisibleCmdlets** parameter to define which *cmdlets* are visible and can be used by the configured role. All cmdlets defined need to be available on the target system to avoid errors.

After creating your **Support.psrc** role capability file, it is also possible to directly edit it using a text editor of your choice. Not only can you use the **-VisibleCmdlets** parameter when creating the role capability file, but you can also configure this option directly in the role capability file.

If you simply want to configure cmdlets without restricting their parameters, you can put them into single quotation marks and separate them with commas. In this example, the configured role would be able to restart services as well as restart the computer:

```
VisibleCmdlets = 'Restart-Service', 'Restart-Computer'
```

When using wildcards to configure cmdlets, it is crucial to be aware of the potential risks involved. While it may seem convenient to use a wildcard to allow a range of commands, you might unintentionally grant more permissions than necessary, creating vulnerabilities in your setup. Using the following command, this role would be able to run all commands that start with **Get-**:

```
VisibleCmdlets = 'Get-*
```

But allowing a role to use all commands that start with **Get-** might also expose sensitive information through the **Get-Content** cmdlet, even if that was not the intended purpose of the role. Therefore, it's important to carefully consider the commands you allow and regularly review and adjust the permissions as needed to maintain the security of your system.

To also restrict the parameters of a cmdlet, you can build simple hash tables, like so:

```
VisibleCmdlets = @{ Name = 'Restart-Service'; Parameters = @{ Name = 'Name'; ValidateSet = 'Dns',
@{ Name = 'Restart-Computer'; Parameters = @{ Name = 'Name'; ValidateSet = 'Confirm', 'Delay', 'Fo
@{ Name = 'Get-ChildItem'; Parameters = @{ Name = 'Path'; ValidatePattern = '^C:\\Users\\[^\\]+$'
```

Using the preceding example, the configured role would be allowed to run three commands: the first allowed cmdlet would be **Restart-Service**, but this role would be only allowed to restart the **dns** and **spooler** services. The second allowed cmdlet empowers the role to also *restart the computer* but only using the **-Confirm**, **-Delay**, **-Force**, and **-Timeout** parameters. And last, but not least, the third allowed cmdlet is **Get-ChildItem**, but with the configuration specified within **ValidatePattern**, a user with this role would only be able to query subfolders of the **C:\Users** path.

VisibleFunctions

VisibleFunctions defines which *functions* are visible and can be used by the configured role. All functions defined need to be either available on the target system or defined in the **FunctionDefinitions** section of the current role capability file to avoid errors.

Functions are defined like cmdlets:

```
VisibleFunctions = 'Restart-NetAdapter'
```

This example would allow the **Restart-NetAdapter** function; if executed, this function restarts a network adapter by disabling and enabling the network adapter again.

For functions, you can also use hash tables to define more complex restrictions and wildcards that also work similarly to cmdlets—these should still be used very carefully.

VisibleAliases

VisibleAliases defines which *aliases* are visible and can be used by the configured role. All aliases defined need to be either available on the target system or defined in the **AliasDefinitions** section of the current role capability file to avoid errors:

```
VisibleAliases = 'cd', 'ls'
```

This example would allow the **cd** alias to allow the **Set-Location** cmdlet and the **ls** alias to allow the **Get-ChildItem** cmdlet.

Aliases are configured in a similar way to cmdlets and functions in the role capability file. Please refer to those sections (*VisibleCmdlets* and *VisibleFunctions*) for further examples.

VisibleExternalCommands

VisibleExternalCommands defines which *traditional Windows executables* are visible and can be used by the configured role. All defined external commands need to be available on the target system to avoid errors. An example of an external command is a standalone executable or an installed program. Always test your configuration to ensure all dependencies are considered by your configuration.

Using this setting, you can allow external commands and PowerShell scripts. Using the following example, you would allow an executable file named **putty.exe**, which is located under **C:\tmp\putty.exe**, as well as a **myOwnScript.ps1** PowerShell script, which can be found under **C:\scripts\myOwnScript.ps1**:

```
VisibleExternalCommands = 'C:\tmp\putty.exe', 'C:\scripts\myOwnScript.ps1'
```

Make sure that you have thoroughly reviewed and tested the script you're exposing and that you have implemented appropriate measures to prevent unauthorized tampering. If you are exposing a script or an executable, always ensure that you have complete control over it and are confident that it will not compromise your configuration.

VisibleProviders

No PowerShell *providers* are available in JEA sessions by default, but by using **VisibleProviders**, you can define which ones are visible and can be used by the configured role. All providers defined need to be available on the target system to avoid errors.

To get a full list of providers, run **Get-PSProvider**, as shown in the following screenshot:

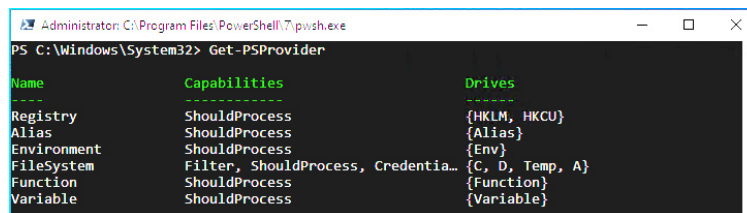


Figure 10.9 – Getting a full list of available providers

For example, if you want to make the **Registry** provider available and, with it, also its **HKEY_LOCAL_MACHINE (HKLM)** and **HKEY_CURRENT_USER (HKCU)** drives, the configuration would look like this:

```
VisibleProviders = 'Registry'
```

Only make providers available if it's really necessary for the role that you are configuring. If this role does not operate with the registry on a regular

basis, consider writing a script or a function instead, if the task is repeatable.

ModulesToImport

Using the **ModulesToImport** parameter, you can define which modules will be imported in the current session. Please note that the modules already need to be installed before they can be imported:

```
ModulesToImport = @{ModuleName='EventList'; ModuleVersion='2.0.2'}
```

Again, it is possible to use hash tables to specify more details. The preceding example would import the **EventList** module in version 2.0.2. Please make sure to use **VisibleFunctions** and/or **VisibleCmdlets** to restrict which functions or cmdlets can be used in this session.

ScriptsToProcess

The specified script file(s) will be executed once the session is established, like a startup script. The path to the script needs to be defined as a full or an absolute path.

The **ScriptsToProcess** parameter allows you to add configured scripts to this role's JEA session, which then will be run in the context of the session. Of course, the script needs to be available on the target system.

The script specified is run as soon as a connection to this session is established:

```
ScriptsToProcess = 'C:\Scripts\MyScript.ps1'
```

If **ScriptsToProcess** is configured for a role within the role capability file, it only applies to this role. If it's configured within a session configuration file, it applies to all roles that are linked to this particular session.

AliasDefinitions

You can use this section to define aliases that were not already defined on the target system and will be only used in the current JEA session:

```
AliasDefinitions = @{Name='ipconfig'; Value='ipconfig'; Description='Displays the Windows IP Configuration'}
VisibleAliases = 'ipconfig'
```

Don't forget to also add the alias to **VisibleAliases** to make it available in the session.

FunctionDefinitions

You can use this section to define functions that are not available on the target system and will be only used in the current JEA session.

If you define a function within **FunctionDefinitions**, make sure to also configure it in the **VisibleFunctions** section:

```

VisibleFunctions = 'Restart-PrintSpooler'
FunctionDefinitions = @{
    Name          = 'Restart-PrintSpooler'
    ScriptBlock = {
        if ((Get-Service -Name 'Spooler').Status -eq 'Running') {
            Write-Warning "Attempting to restart Spooler service..."
            Restart-Service -Name 'Spooler'
        }
        else {
            Write-Warning "Attempting to start Spooler service..."
            Start-Service -Name 'Spooler'
        }
    }
}

```

This example creates a **Restart-PrintSpooler** custom function that first checks if the spooler service is running. If it's running, it will be restarted, and if it's not running, it will attempt to be started.

If you are referring to other modules, use the **fully qualified module name (FQMN)** instead of aliases.

Instead of writing a lot of custom functions, it may be easier to write a PowerShell script module and configure **VisibleFunctions** and **ModulesToImport**.

VariableDefinitions

You can use this section to define variables that will be only used in the current JEA session. Variables are defined within a hash table. Variables can be statically or dynamically set:

```

VariableDefinitions = @{ Name = 'Variable1'; Value = { 'Dynamic' + 'InitialValue' } }, @{ Name = '

```

The following code line is an example of a static variable and a dynamic variable set using **VariableDefinitions**:

```

VariableDefinitions =@{TestShare1 = '$Env:TEMP\TestShare'; TestShare2 = 'C:\tmp\TestShare'}

```

Two variables would be defined in this example: while the first variable, **\$TestShare1**, is dynamically set and refers to where the **\$Env:TEMP** environment variable would lead to, the second one, **\$TestShare2**, is static and would always point to **'C:\tmp\TestShare'**.

Variables can also have options configured. This parameter is optional and is **None** by default. Acceptable parameters are **None**, **ReadOnly**, **Constant**, **Private**, or **AllScope**.

EnvironmentVariables

You can use this section to define environment variables that will be only used in the current JEA session. Environment variables are defined as hash tables:

```

EnvironmentVariables = @{Path= '%SYSTEMROOT%\system32; %SYSTEMROOT%\System32\WindowsPowerShell\v1.

```

The previous example would set the environment **Path** environment variable to contain the '%SYSTEMROOT%\system32; %SYSTEMROOT%\System32\WindowsPowerShell\v1.0\;C:\Program Files\PowerShell\7\;C:\Program Files\Git\cmd' string to enable programs such as Windows PowerShell, PowerShell 7, and Git to find their executables and run without prompting the user.

TypesToProcess

You can use **TypesToProcess** to specify **types.ps1xml** files that should be added to the configured session. Type files are usually specified as **.ps1xml** files. Use the full or absolute path to define type files in the role capability file:

```
TypesToProcess = 'C:\tmp\CustomFileTypes.ps1xml'
```

You can find more information about type files in the official documentation:

https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_types.ps1xml

FormatsToProcess

You can use the **FormatsToProcess** parameter to specify which formatting files should be loaded in the current session. Similar to type files, formatting files are also configured within files that end with **.ps1xml**. Also, for **FormatsToProcess**, the path must be specified as a full or an absolute path:

```
FormatsToProcess = 'C:\tmp\CustomFormatFile.ps1xml'
```

You can find more information about formatting files in the official documentation:

https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_format.ps1xml

AssembliesToLoad

To make the types contained in binary files available for the scripts and functions that you write, use the **AssembliesToLoad** parameter to specify the desired assemblies. This enables you to leverage the functionality provided by these assemblies in the JEA session:

```
AssembliesToLoad = "System.Web", "FSharp.Compiler.CodeDom.dll", 'System.OtherAssembly, Version=4.0.
```

If you want to learn more about role capability files in JEA and more options such as creating custom functions especially for one role, please refer to the official documentation:

<https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/role-capabilities>

If you want a role capability file to be updated, you can do this at any time by simply saving changes to the role capability file. Any new JEA session that is established after the changes were made will represent the updated changes.

Role capabilities can also be merged when a user is granted access to multiple role capabilities. Please refer to the official documentation to learn which permissions will be applied in this case:

<https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/role-capabilities#how-role-capabilities-are-merged>

Now that you have defined your roles—for a better overview, create each one in a separate role capability file—it's time to assign them to certain users and groups and define session-specific parameters. This can be done within a session configuration file.

Session configuration file

Using a session configuration file, you can specify who is allowed to connect to which endpoint. Not only can you map users and groups to specific roles, but you can also configure global session settings such as which scripts should be executed when connected to the session, logging policies, or which identity will be used when you connect (for example, virtual accounts or **group Managed Service Accounts (gMSAs)**). If you want to, you can configure session files on a per-machine basis.

You can create a skeleton session configuration file using the **New-PSSessionConfigurationFile** cmdlet:

```
New-PSSessionConfigurationFile -Path .\JEA_SessionConfigurationFile.pssc
```

Similar to creating a skeleton role capability file, a prepopulated session configuration file that can be edited is created:

```
1 @{
2
3   # Version number of the schema used for this document
4   SchemaVersion = '2.0.0.0'
5
6   # ID used to uniquely identify this document
7   GUID = '45e275e-ca82-4d17-89be-216f74b6a52e'
8
9   # Author of this document
10  Author = 'Miriam Wiesner'
11
12  # Description of the functionality provided by these settings
13  # Description = ''
14
15  # Session type defaults to apply for this session configuration. Can be 'RestrictedRemoteServer' (recommended), 'Empty', or 'Default'
16  SessionType = 'Default'
17
18  # Directory to place session transcripts for this session configuration
19  # TranscriptDirectory = 'C:\Transcripts\'
20
21  # Whether to run this session configuration as the machine's (virtual) administrator account
22  # RunAsVirtualAccount = $true
23
24  # Scripts to run when applied to a session
25  # ScriptsToProcess = 'C:\ConfigData\InitScript1.ps1', 'C:\ConfigData\InitScript2.ps1'
26
27  # User roles (security groups), and the role capabilities that should be applied to them when applied to a session
28  # RoleDefinitions = @( 'CONTOSO\SqlAdmins' - @('RoleCapabilities - 'SqlAdministration' ); 'CONTOSO\SqlManaged' - @('RoleCapabilities - '
29
30 }
```

Figure 10.10 – An empty skeleton session configuration file

In the session configuration file, there are again some *general parameters* that help you describe this file. Some of them are listed here:

- **SchemaVersion:** Describes the schema version number of this document, which is usually **2.0.0.0**, if not specified otherwise.
- **GUID:** A GUID is a unique, randomly generated UID to identify this file.
- **Author:** The author who created this document.
- **Description:** A description of this session configuration file. It makes sense to be specific so that you can easily edit and operate your base of growing configuration files.

Let's look at which other options you can configure using the session configuration file.

Session type

The session type indicates what kind of session is created (language mode-wise) and which commands are allowed. For a JEA session configuration file, you should *always configure* **SessionType = 'RestrictedRemoteServer'**.

In regular session files, you can use the following values for this parameter:

- **Default:** This configuration provides an *unrestricted PowerShell endpoint*. This means that users can run *any* command that is available on the system. It is not recommended to use this session type when configuring JEA.
- **Empty:** No modules and no commands are added to the session. Only if you had configured **VisibleCmdlets**, **VisibleFunctions**, and other parameters in the session configuration file would your session be populated. Don't use these settings when configuring JEA, unless you have a use case to even restrict the cmdlets that are allowed when configuring **RestrictedRemoteServer**.
- **RestrictedRemoteServer:** This value should be used when creating a JEA session configuration file. It appropriately limits the language mode and only imports a small set of essential commands, such as **Exit-PSSession**, **Get-Command**, **Get-FormatData**, **Get-Help**, **Measure-Object**, **Out-Default**, and **Select-Object**, which are sufficient for most administrative tasks. This configuration provides a higher level of security as it restricts access to potentially dangerous cmdlets and functions.

When creating the base session configuration file, you can use the **SessionType** parameter to directly configure the session type, like so:

```
> New-PSSessionConfigurationFile -SessionType RestrictedRemoteServer -Path .\JEA_SessionConfigurat
```

TranscriptDirectory

Session transcripts record all commands that are being run in a particular session, as well as the output. It is recommended to use session transcripts for every user and audit which commands are being executed. This can be achieved by using the **TranscriptDirectory** parameter.

First, make sure to preconfigure a folder on the JEA endpoint to store the transcripts. This folder needs to be a protected folder so that regular users cannot modify or delete any data within this folder. Also, make sure that the local system account is configured to have read and write access, as this account will be used to create transcript files.

In the best case, also make sure that the transcript files are regularly uploaded and parsed to your **Security Information and Event Management (SIEM)** system so that they are in a central location. Also, make sure to implement a mechanism to rotate log files so that the hard disk does not run out of space.

Everything set up? Good! Now, it's time to configure the path to the pre-configured folder in the session configuration file, as follows:

```
TranscriptDirectory = 'C:\Jea-Transcripts'
```

Using the preceding configuration would write all transcripts to the **C:\Jea-Transcripts** folder. New files will always be generated using a timestamp so that no file is overwritten.

Additional to the **TranscriptDirectory** parameter, also make sure to implement proper auditing. See [Chapter 4, Detection – Auditing and Monitoring](#), for more details.

Configuring the JEA identity

When using JEA, you don't use your regular account on the target system. But which account will be used instead?

With JEA, there are two possibilities when it comes to identities: using either a **virtual account** or a **gMSA**. Using a virtual account is the method that you should always prefer unless you need access to network resources during the JEA session. In the following sections, we will learn more about both options and explore why a virtual account is a more secure option.

Virtual account

When in doubt, configuring a virtual account should always be your preferred option. A virtual account is a temporary administrator account that is created at the start of a JEA session and is destroyed once the session ends. This means that it only lasts for the duration of the remote session, making it a secure option for providing temporary administrative access. A huge advantage is that at no point in time do reusable credentials enter the system.

When connecting to an endpoint, the non-administrator user connects and runs all commands in the session as a privileged virtual account. This account is a local administrator or a domain administrator account on **domain controllers (DCs)** but nevertheless is restricted to running only the commands that are allowed for this role.

To follow this example more easily, I have created a simple script to create a **ServerOperator** role and register it together with a session configuration that lets the connecting user connect as a virtual account. Let's use this configuration to demonstrate the examples within this chapter.

You can find the script in the GitHub repository of this book under <https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter10/JEA-ServerOperator.ps1>.

In my example, I execute all commands on **PSec-Srv01**, a Windows 2019 server that was joined to the **PSec** domain.

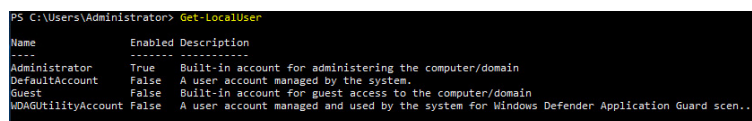
First, make sure that the account you want to configure the **ServerOperator** role for is present in your environment, and possibly adjust the username in the script. In my demo example, the user is **PSec\mwiesner**.

Then, run the **JEA-ServerOperator.ps1** script from the GitHub repository to ensure that the **ServerOperator** JEA endpoint was created successfully.

Once the endpoint has been successfully created, establish a session to the localhost, using the **ServerOperator** JEA session:

```
> Enter-PSSession -ComputerName localhost -ConfigurationName ServerOperator -Credential $ServerOpe
```

Once a JEA session is established that relies on a virtual account, let's check the actual local user accounts by running the **Get-LocalUser** command from a separate elevated PowerShell console. As you can see, there was no additional local account created for the JEA connection:



```
PS C:\Users\Administrator> Get-LocalUser
Name           Enabled Description
-----
Administrator   True    Built-in account for administering the computer/domain
DefaultAccount  False   A user account managed by the system.
Guest            False   Built-in account for guest access to the computer/domain
WDAGUtilityAccount False   A user account managed and used by the system for Windows Defender Application Guard scen...
```

Figure 10.11 – No additional local account was created

To verify which virtual accounts are or were signed in during the current uptime of the machine, I have written a script to help you see which virtual accounts were created for your JEA sessions:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter10/Get-VirtualAccountLogons.ps1>

The script uses **Get-CimInstance** to retrieve information about logged-on users and their logon sessions, merges the information, and displays which virtual accounts were created and whether the session is still active or inactive.

The following screenshot shows you the output of the **Get-VirtualAccountLogons.ps1** script:

```

PS C:\Users\Administrator\Documents> .\Get-VirtualAccountLogons.ps1

Name           : WinRM VA_1_PSSEC_mwiesner
Domain         : WinRM Virtual Users
LogonTypeNumber : 5
LogonTypeString : Service
SessionStartTime : 10/04/2023 21:39:37
SessionAuthPackage : Negotiate
LogonId        : 2639931
ActiveSession   : False

Name           : WinRM VA_2_PSSEC_mwiesner
Domain         : PSSEC-PC01
LogonTypeNumber : 5
LogonTypeString : Service
SessionStartTime : 10/04/2023 21:39:56
SessionAuthPackage : Negotiate
LogonId        : 2770910
ActiveSession   : True

```

Figure 10.12 – Virtual account usage for the current uptime can be assessed using the Get-VirtualAccountLogons.ps1 script

All virtual accounts that were created until the operating system reboots are cached in the **Common Information Model (CIM)** tables, therefore you can see past as well as current virtual account connections. If the session is still established, the script indicates it with **ActiveSession: True**.

All virtual account names that are generated through an established JEA session follow the "WinRM VA_<number>_<domain>_<username>" scheme. If multiple sessions from the same user account were to be established, the number would be raised accordingly.

DID YOU KNOW?

Retrieving a list of all current users is also possible by using the deprecated `Get-WmiObject win32_process).GetOwner().User` Windows Management Instrumentation (WMI) command.

Therefore, if you don't need to access network resources, the best option to configure the identity of your JEA session is to use a virtual account.

gMSA

If you need to access network resources (for example, other servers or network shares), a gMSA is an alternative to a virtual account.

You can find more information on how to create and configure a gMSA in the official documentation:

<https://docs.microsoft.com/en-us/windows-server/security/group-managed-service-accounts/group-managed-service-accounts-overview>

You can use a gMSA account to authenticate against your domain and therefore access resources on any domain-joined machine. The rights a user gets by using a gMSA account are determined by the resources that will be accessed. Only if a gMSA account was explicitly granted admin privileges causes the user using the gMSA account have administrator rights.

A gMSA is an account that is managed by Active Directory and changes its password on a frequent basis. As such, the password could be reused by

an adversary—if captured—but only for a limited time.

In the best case, use a virtual account; only use gMSA accounts when your tasks require access to network resources for some particular reasons, such as the following:

- It is more difficult to determine who performed which actions under the identity of a gMSA as the same account is used by every user connecting to a session with the same gMSA account. To determine which user performed which action, you would need to correlate PowerShell session transcript files with the according events from event logs.
- There is a possibility to grant more rights than the JEA configuration plans to, as a gMSA account might have access to many network resources that are not needed. Always follow the least-privilege principle to restrict your JEA sessions effectively.

gMSAs are only available starting from Windows PowerShell 5.1 or higher and can only be used on domain-joined machines. Of course, it is also possible to use a standalone domain if you don't want to join the machine to your production domain.

Choosing the JEA identity

Once you have chosen the identity you want to use to connect to your JEA session, it's time to configure it. You will need to configure either a virtual account or a gMSA, and you can do this in your JEA session configuration file.

Configure a local virtual account using the following options:

- `RunAsVirtualAccount = $true`
- `RunAsVirtualAccountGroups = 'NetworkOperator', 'NetworkAuditor'`

Using the `RunVirtualAccountGroups` parameter, you can define in which groups the virtual account should reside. To prevent the virtual account from being added to the local or domain administrators group by default, you will need to specify one or more security groups.

Define a gMSA using the `GroupManagedServiceAccount` parameter, like so:

```
GroupManagedServiceAccount = 'MyJEAgMSA'
```

Also, refer to the official session configuration documentation:

<https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/session-configurations>

ScriptsToProcess

Similar to `ScriptsToProcess`, which can be configured within the role capability file. See the *ScriptsToProcess* subsection of the section entitled *The role capability file* to learn more about it and how to configure it.

If **ScriptsToProcess** is configured for a role within the role capability file, it only applies to this role. If it's configured within a session configuration file, it applies to all roles that are linked to this particular session.

RoleDefinitions

Role definitions connect the roles that you have configured in the role capability file with the current session configuration file and can be configured within a hash table, like so:

```
RoleDefinitions = @{
    'CONTOSO\JEA_DNS_ADMINS' = @{ RoleCapabilities = 'DnsAdmin', 'DnsOperator', 'DnsAuditor' }
    'CONTOSO\JEA_DNS_OPERATORS' = @{ RoleCapabilities = 'DnsOperator', 'DnsAuditor' }
    'CONTOSO\JEA_DNS_AUDITORS' = @{ RoleCapabilities = 'DnsAuditor' }
}
```

You can assign one or more role capabilities to a user account or to an Active Directory group.

Conditional access

JEA itself is already a great option to restrict the exact commands a role is allowed to execute on an endpoint, but all users or groups that are assigned a role are able to run the configured commands. But what if you want to set up more restrictions, such as—for example—enforcing the users to also use **multi-factor authentication (MFA)**?

This is where additional access comes into play. Using the **RequiredGroups** parameter, you can enforce that connecting users are part of a defined group—a group that you can use to enforce more conditions on the user.

Using **And** or **Or** helps you to define more granular rules.

Using the following example, all connecting users must belong to a security group named **MFA-logon**; simply use the **And** condition:

```
RequiredGroups = @{ And = 'MFA-logon' }
```

Sometimes, you have different ways to authenticate or to provide additional security. So, if you want those connecting users to be either in the **MFA-logon** *OR* **smartcard-logon** group, use the **Or** condition, as shown in the following example:

```
RequiredGroups = @{ Or = 'MFA-logon', 'smartcard-logon' }
```

Of course, you can also create more complicated, nested conditions by combining **And** and **Or** conditions.

In the following example, connecting users need to be part of the **elevated-jea** group and need to be either logged in with MFA or a smart card:

```
RequiredGroups = @{ And = 'elevated-jea', @{ Or = 'MFA-logon', 'smartcard-logon' } }
```

However, regardless of which configuration option(s) you use, always make sure to test that your conditions are applied as planned.

User drive

It is possible to copy files from a JEA session remotely by configuring and leveraging a user drive. For example, you can copy log files from your session for detailed analysis later on your normal work computer.

To configure a user drive with a capacity of 10 MB, use the following configuration:

```
MountUserDrive = $true
UserDriveMaximumSize = 10485760
```

After accessing a JEA session that has a user drive configured, you can easily copy files from or to the session.

The following example shows how to copy the **myFile.txt** file into your **\$ServerOperator** JEA session:

```
Copy-Item -Path .\myFile.txt -Destination User: -ToSession $ServerOperator
```

The next example shows how to copy the **access.log** file from the remote machine within the **\$ServerOperator** JEA session to your local one:

```
Copy-Item -Path User:\access.log -Destination . -FromSession $jeasession
```

Although you can copy files from and into the established JEA session, it is not possible to specify the filename or subfolder on the remote machine.

If you want to learn more about PowerShell drives, also have a look at

<https://docs.microsoft.com/en-us/powershell/scripting/samples/managing-windows-powershell-drives>.

Access rights (SDDL)

Access rights to the JEA session are configured per SDDL.

So when you are using JEA, SDDLs will get configured automatically when assigning user/group **Access Control Lists (ACL)** to a session configuration. The group and the **Security Identifier (SID)** will be both looked up and automatically added with the appropriate level of access to the session configuration.

You can find out the SDDL of a session configuration by running the **(Get-PSSessionConfiguration -Name <session configuration name>).SecurityDescriptorSddl** command:

```
PS C:\Users\Administrator> (Get-PSSessionConfiguration -Name ServerOperator).SecurityDescriptorSddl
O:NSG:BAD:P(A;;GA;;;S-1-5-21-3035173261-3546990356-1292108877-1601)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)
PS C:\Users\Administrator>
```

Figure 10.13 – Finding the SDDL of a session configuration

Refer to the official documentation to learn more about the SDDL syntax:

<https://docs.microsoft.com/en-us/windows/desktop/secauthz/security-descriptor-definition-language>

Deploying JEA

To deploy JEA, you need to understand which commands the users you want to restrict are using. If you ask me, this is the hardest part about JEA.

But there are tools, such as my self-written JEAAnalyzer open source project, that ease this task massively. I will come back to this tool later in this chapter.

Once you have identified the commands used and the users and groups you want to restrict, first create a session capability file and a role capability file. The following diagram shows the steps you will need to take to deploy JEA:

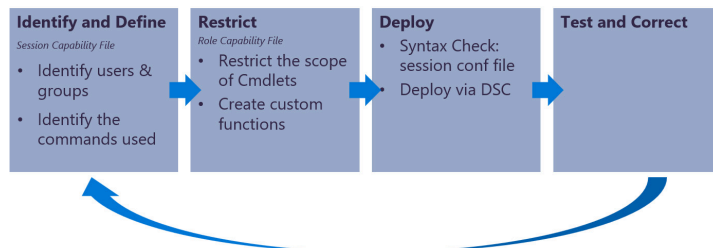


Figure 10.14 – Steps to deploy JEA

Once you have created both required files, make sure to check the syntax of the session configuration file before deploying the files using the **Test-PSSessionConfigurationFile -Path <path to session configuration file>** cmdlet.

If a JEA session configuration needs to be changed—for example, to map or remove users to or from a role—you will always need to unregister and register the JEA session configuration again. If you only want to change roles configured in the role capability file, it is enough to simply change the configuration; there's no need to re-register the session configuration.

You can also verify which capabilities a specific user would get in a specific session by running **Get-PSSessionCapability -ConfigurationName <configuration name> -Username <username>**.

Once you are ready to deploy, you will need to decide which deployment mechanism you will use. There's the option to either *register the session manually* or to use **Desired State Configuration (DSC)** for the deployment.

Registering manually

Registering the machine manually is a great option if you just want to test your configuration on a few machines or if you only need to administer

small environments. Of course, you can also script the deployment process using manual registration commands, but you still need to find a way to deploy your scripts.

Therefore for big environments, DSC might be the better solution for you.

Before registering manually, ensure that at least one role was added to the **RoleCapabilities** file and that you created and tested the accompanying session configuration file.

In order to register your JEA configuration successfully, you will need to be a local administrator on the system(s).

If everything is in place, adjust the following command to your configuration and run it on the endpoint to configure:

```
Register-PSSessionConfiguration -Path .\MyJEAConfig.pssc -Name 'JEA Maintenance' -Force
```

After registering a session, make sure to restart the WinRM service to ensure that the new session is loaded and active:

```
Restart-Service -name WinRM
```

For a working example, refer to the demo configuration file on this book's GitHub repository:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter10/JEA-ServerOperator.ps1>

Deploying via DSC

In big environments, it might be worthwhile to leverage DSC. DSC is a really cool way to tell your remote servers to “make it so” and to apply your chosen configuration regularly.

Even if someone were to change the configuration on the server, with DSC configured, your servers could reset themselves without any intervention from an administrator, as they can pull and adjust their own configuration to your configured baseline on a frequent basis.

DSC is a big topic, therefore I cannot describe the entire technique in detail, but if you want to learn more about it, review *Chapter 13. What Else? – Further Mitigations and Resources*, and have a look at the official documentation.

For a basic JEA DSC configuration, please refer to the *Registering JEA Configurations* documentation:

<https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/register-jea?#multi-machine-configuration-with-dsc>

Connecting to the session

Once you have set up your JEA sessions, make sure that users that should connect to the JEA sessions have the **Access this computer from the network** user right configured.

Now is the big moment, and you can connect to the JEA session:

```
Enter-PSSession -ComputerName <computer> -ConfigurationName <configuration name>
```

By default, it is not possible to use *Tab* to autocomplete commands on the command line. If you want to have it accessible, nevertheless, it is recommended to use **Import-PSSession**, which allows features such as *Tab* completion to work without impacting security:

```
> $jeasession = New-PSSession -ComputerName <computer> -ConfigurationName <configuration name>  
> Import-PSSession -Session $jeasession -AllowClobber
```

It is recommended to *not* configure the **TabExpansion2** function as a visible function, as this executes all kinds of code and is dangerous for the security of your secure environment.

To display all available session configurations on the local machine, run **Get-PSSessionConfiguration**.

Once you have successfully configured, deployed, and tested your JEA sessions, make sure to remove all other access possibilities for the connecting user. Even if you have the best JEA configuration deployed, it's worth nothing if your users can bypass it by leveraging another connection possibility—for example, by connecting over Remote Desktop.

Deploying JEA seems like a bunch of work to get it running at first glance, right? But don't worry—there are actually ways that can simplify your work, such as JEAnalyzer.

Simplifying your deployment using JEAnalyzer

When I first learned about JEA, I evangelized it and told everyone how awesome this solution was. Isn't it awesome restricting the commands your users are allowed to run to exactly to what is needed? Isn't it amazing to configure virtual accounts and completely avoid passing the hash when using JEA and virtual accounts?

Yes, it is! But when I talked to customers about JEA and how awesome it was, I quickly received the same questions over and over again: *How can we find out which commands our users and administrators are using? How can we create those role capability files in the easiest way?*

And this was the time when I had the idea for the JEAnalyzer module. After I started the project, my friend Friedrich Weinmann was also very interested in this project, and when I switched jobs and barely worked with customers on other topics than Microsoft Defender for Endpoint, I was glad that he took over what I started and maintained the repository and included our remaining common visions for the project.

You can find the JEAnalyzer repository on GitHub:

<https://github.com/PSecTools/JEAnalyzer>

JEAnalyzer is a PowerShell module that can be easily installed over the PowerShell Gallery, using the **Install-Module JEAnalyzer -Force** command. After agreeing to all popups, provided by NuGet and others, the module will be installed and can be imported using **Import-Module JEAnalyzer**.

At the time this book was written, the latest version of JEAnalyzer was **1.2.10** and consists of 13 functions, as illustrated here:

```
PS C:\Users\Administrator> Get-Command -module jeanalyzer
```

CommandType	Name	Version	Source
Function	Add-JeaModuleRole	1.2.10	jeanalyzer
Function	Add-JeaModuleScript	1.2.10	jeanalyzer
Function	ConvertTo-JeaCapability	1.2.10	jeanalyzer
Function	Export-JeaModule	1.2.10	jeanalyzer
Function	Export-JeaRoleCapFile	1.2.10	jeanalyzer
Function	Import-JeaScriptFile	1.2.10	jeanalyzer
Function	Install-JeaModule	1.2.10	jeanalyzer
Function	New-JeaCommand	1.2.10	jeanalyzer
Function	New-JeaModule	1.2.10	jeanalyzer
Function	New-JeaRole	1.2.10	jeanalyzer
Function	Read-JeaScriptblock	1.2.10	jeanalyzer
Function	Read-JeaScriptFile	1.2.10	jeanalyzer
Function	Test-JeaCommand	1.2.10	jeanalyzer

Figure 10.15 – Available functions of JEAnalyzer

Every function is very well documented so I will not describe all functions, just the most important ones to find out which commands your users are using and how to simply create your first role capability and session configuration files with the help of JEAnalyzer.

Converting script files to a JEA configuration

If you have a certain script logic that needs to be run within a JEA session and simply want to convert the script into an endpoint configuration, JEAnalyzer has you covered.

As a demo script file, I used the **Export AD Users to CSV** script that was originally written by Victor Ashiedu in 2014. You can find a version of this script here:

https://github.com/sacroucher/ADScripts/blob/master/Export_AD_Users_to_CSV.v1.0.ps1

Download the script and save it under

C:\DEMO\ext\Export_AD_Users_to_CSV.v1.0\Export_AD_Users_to_CSV.v1.0.ps1.

Also, create a folder under **C:\JEA** to store the output files.

After you are well prepared, download the script from this book's GitHub repository and make sure to follow it command after command. Don't run it as a whole script to make sure that you understand every single step—the script is well commented.

You can find the script under

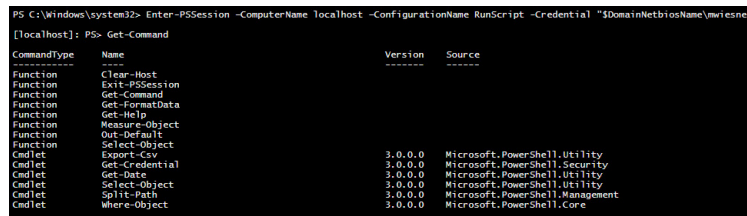
<https://github.com/PacktPublishing/PowerShell-Automation-and->

[Scripting-for-Cybersecurity/blob/master/Chapter10/JEAAnalyzer-AnalyzeScripts.ps1](https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter10/JEAAnalyzer-AnalyzeScripts.ps1).

The most important commands used from JEAAnalyzer for this example are outlined here:

- **Read-JeaScriptFile:** Parses and analyzes a script file for qualified commands. Make sure to specify the script using the **-Path** parameter.
- **Export-JeaRoleCapFile:** Converts a list of commands into a JEA role capability file.

After entering the newly created session and analyzing the commands configured, you can see that all the commands used, as well as the standard session functions, are allowed within this session:



CommandType	Name	Version	Source
Function	Clear-Host		
Function	Exit-PSsession		
Function	Get-Command		
Function	Get-FormatData		
Function	Get-Help		
Function	Measure-Object		
Function	Out-Default		
Function	Select-Object		
Cmdlet	Export-Csv	3.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Get-Credential	3.0.0.0	Microsoft.PowerShell.Security
Cmdlet	Get-Date	3.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Select-Object	3.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Split-Path	3.0.0.0	Microsoft.PowerShell.Management
Cmdlet	Where-Object	3.0.0.0	Microsoft.PowerShell.Core

Figure 10.16 – Displaying all allowed functions and commands

But sometimes, auditing and configuring only script files is not enough; sometimes you also need to configure sessions for your users and administrators, allowing commonly used commands and functions.

Using auditing to create your initial JEA configuration

To follow this example, you will need this section's script from the GitHub repository:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter10/JEAAnalyzer-AnalyzeLogs.ps1>

Similar to the demo script from the converting script files, don't run this script in its entirety, but make sure to follow it command by command to understand the examples.

As a prerequisite, make sure to install the **ScriptBlockLoggingAnalyzer** module, which was created by Dr. Tobias Weltner:

```
> Install-Module ScriptBlockLoggingAnalyzer
```

Also, before we can leverage auditing, we need to enable **ScriptBlockLogging**. Therefore either enable **ScriptBlockLogging** manually on your local machine or make sure to enable it for multiple machines. Refer to [Chapter 4, Detection – Auditing and Monitoring](#), to learn more about **ScriptBlockLogging**.

Using these commands, you can enable **ScriptBlockLogging** manually on your local machine, like so:

```
> New-Item -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging" -Force
> Set-ItemProperty -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"
```

At some point in the script, you will be asked to run some commands as another user. In my demo environment, I run the commands as the **mwiesner** user. If you configured another user for your demo purposes, make sure to run this session under your customized user account and adjust the script accordingly.

To run commands as **mwiesner** or another user, right-click on the PowerShell console and select **Run as different user**. Depending on the configuration of your system, it might be necessary to press *Shift* and then right-click on the PowerShell console to make this option appear.

Run some demo commands in this session. You can find some examples in the script. Just make sure to run one command after the other, and don't run it as one big script block.

Then, follow the script's examples, analyze the commands, and create an initial JEA configuration out of the audited commands. The most important commands used in this script are set out here:

- **Get-SBLEvent** (**ScriptBlockLoggingAnalyzer** module): Reads **ScriptBlockLogging** events from the PowerShell audit log
- **Read-JeaScriptblock**: Parses and analyzes passed code for qualified commands, when specified using the **-ScriptCode** parameter
- **Export-JeaRoleCapFile**: Converts a list of commands into a JEA role capability file

Use the script to explore how to create an initial JEA session out of audited commands and adjust the commands used to your needs. In this way, it will be easy to create an initial JEA role capability file to adjust and fine-grain later.

But also once you have started using JEA, auditing is quite important within your JEA sessions. Let's look in the next section at how you can leverage it and link important events related to your users' JEA sessions.

Logging within JEA sessions

When using JEA, logging is of course possible, and you also should implement it and regularly review audit logs to make sure your JEA configuration is not abused in an unforeseen way.

We already covered logging extensively in [Chapter 4, Detection – Auditing and Monitoring](#), therefore here's only a little summary of what's important for logging when it comes to JEA.

Over-the-shoulder transcription

Always configure over-the-shoulder transcription for users running commands via a JEA session. Over-the-shoulder transcription can be configured within the *session configuration file* using the **TranscriptDirectory** parameter, as we discussed earlier in the *TranscriptDirectory* section.

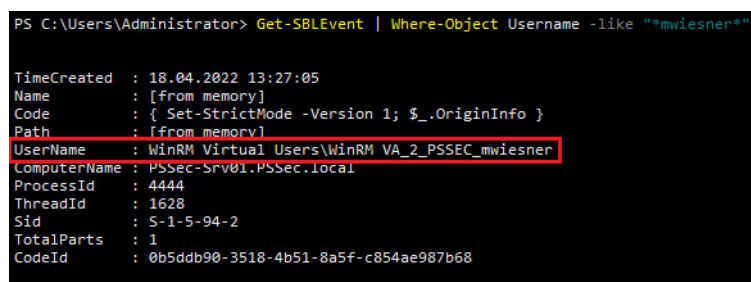
Make sure to protect the configured folder so that its contents cannot be manipulated by an adversary. Also forward, parse, and review the transcripts regularly.

Over-the-shoulder transcription records contain information about the user, the virtual user, the commands that were run in the session, and more.

PowerShell event logs

Not only for finding out who runs which commands, PowerShell event logs are quite useful; when Script Block Logging is turned on, all PowerShell actions are also recorded in regular Windows event logs.

Enable Script Block Logging as well as Module Logging and look for event ID **4104** in the *PowerShell operational log*. On the remote machine, the user you will need to look for is the WinRM virtual user if a virtual account is used. If a *gMSA* account was used, make sure to also watch out for this account. The following screenshot shows a Script Block Logging event for a virtual account:



```
PS C:\Users\Administrator> Get-SBEvent | Where-Object Username -like "*mwiesner*"

TimeCreated : 18.04.2022 13:27:05
Name        : [from memory]
Code        : { Set-StrictMode -Version 1; $_.OriginInfo }
Path        : [from memory]
Username     : WinRM Virtual Users\WinRM VA_2_PSSEC_mwiesner
ComputerName : PSSEC-Srv01.PSSEC.local
ProcessId    : 4444
ThreadId     : 1628
Sid          : S-1-5-94-2
TotalParts   : 1
CodeId       : 0b5ddb90-3518-4b51-8a5f-c854ae987b68
```

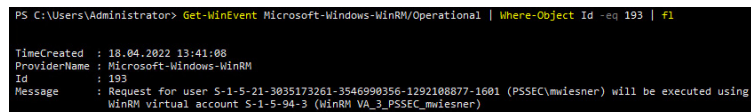
Figure 10.17 – Virtual account is shown as username

Monitor especially for event IDs 4100, 4103, and 4104 in the PowerShell operational log. On some occasions, you will see that the connecting user is the actual user, while the user specified is the WinRM virtual account.

Other event logs

Unlike PowerShell operational logs and transcripts, other logging mechanisms will not capture the connected user. To find out which users connected at which time, you need to correlate event logs.

To do so, look for event ID **193** in the *WinRM operational log* to find out which virtual account or *gMSA* was requested by which user:



```
PS C:\Users\Administrator> Get-WinEvent Microsoft-Windows-WinRM/Operational | Where-Object Id -eq 193 | fl

TimeCreated : 18.04.2022 13:41:08
ProviderName : Microsoft-Windows-WinRM
Id           : 193
Message      : Request for user S-1-5-21-3035173261-3546998356-1292108877-1601 (PSSEC\mwiesner) will be executed using WinRM virtual account S-1-5-94-3 (WinRM VA_3_PSSEC_mwiesner)
```

Figure 10.18 – Using the WinRM operational log for correlation

You can also get more details out of the security log by looking for event IDs **4624** and **4625**. In the following example screenshot, we are looking at two events with the ID **4624** (An account was successfully logged on.) that were generated at the same time—one shows a regular account logon while the other shows the logon of the virtual account:

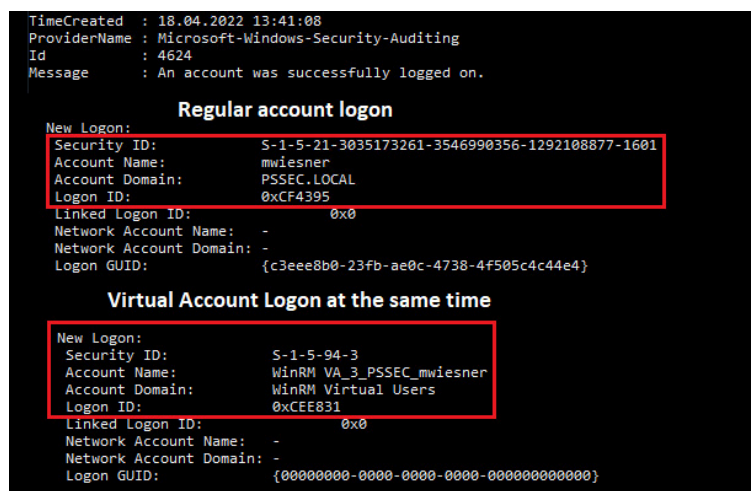


Figure 10.19 – Comparing the regular account and the virtual account logon

If you are looking for more activities in other event logs, use the **Logon ID** value to correlate activities to identified logon sessions.

An account logoff can be identified by event **4634**. Refer to [Chapter 4, Detection – Auditing and Monitoring](#), for more information about the Windows event log.

Best practices – avoiding risks and possible bypasses

JEA is a great option to harden your environment and allow administrators and users to only execute the commands that they need for their daily work. But as with every other technology, JEA can also be misconfigured, and there are risks that you need to watch out for.

Do not grant the connecting user admin privileges to bypass JEA—for example, allowing commands to edit admin groups such as **Add-ADGroupMember**, **Add-LocalGroupMember**, **net.exe**, and **dsadd.exe**. Rogue administrators or accounts that were compromised could easily escalate their privileges.

Also, don't allow users to run arbitrary code, such as malware, exploits, or custom scripts to bypass protections. Commands that you should especially watch out for are (not exclusively) **Start-Process**, **New-Service**, **Invoke-Item**, **Invoke-WmiMethod**, **Invoke-CimMethod**, **Invoke-Expression**, **Invoke-Command**, **New-ScheduledTask**, **Register-ScheduledJob**, and many more.

If your admins really need one of those risky commands, you can try to fine-grain the configuration by also configuring dedicated parameters or by creating and allowing a custom function.

Try to avoid wildcard configurations as they could be tampered with, and be careful when using tools that help you to create a configuration; always review and test the configuration carefully before using it in production.

To protect your role capability and session configuration files from being tampered with, use signing. Make sure to implement a proper logging mechanism and secure transcript files as well as event logs. Also, review them on a regular basis.

And last but not least, when going live, be aware that none of this matters if you do not take away admin rights and remote desktop access to the servers!

Summary

In this chapter, you have learned what language modes are and how they differ from JEA. You have also learned what JEA is and how to set it up.

You now know which parameters you can use to create your own customized JEA role capability and session configuration files (or at least where to go in the book to look for them) and how to register and deploy your JEA endpoints.

Following the examples from this book's GitHub repository, you have managed to create and explore your own JEA sessions, and you have been provided with an option on how to create a simple first configuration out of your own environment, using JEAAnalyzer. Of course, you will still need to fine-tune your configuration, but the first step is done easily.

You have explored how to interpret logging files to correlate JEA sessions over different event logs and what kinds of risks to look out for when creating your JEA configurations.

JEA is a great step to define which commands can be executed by which role, but sometimes you might want to completely prohibit a certain application or just whitelist allowed applications and scripts in your environment. In our next chapter, we will discover how this goal can be achieved using AppLocker, Application Control, and script signing.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, follow these resources:

- PowerShell Constrained Language Mode:
<https://devblogs.microsoft.com/powershell/powershell-constrained-language-mode/>

- about_Language_Modes: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes
- Just Enough Administration (official Microsoft documentation): <https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/overview>
- JEAAnalyzer on GitHub: <https://github.com/PSecTools/JEAAnalyzer>
- PowerShell ♥ the Blue Team: <https://devblogs.microsoft.com/powershell/powershell-the-blue-team/>

You can also find all links mentioned in this chapter in the GitHub repository for [*Chapter 10*](#)—there's no need to manually type in every link:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter10/Links.md>