JEA Role Capabilities

07/10/201912 minutes to read  

In this article

When creating a JEA endpoint, you need to define one or more role capabilities that describe what someone can do in a JEA session. A role capability is a PowerShell data file with the `.psrc` extension that lists all the cmdlets, functions, providers, and external programs that are made available to connecting users.

Determine which commands to allow

The first step in creating a role capability file is to consider what the users need access to. The requirements gathering process can take a while, but it's an important process. Giving users access to too few cmdlets and functions can prevent them from getting their job done. Allowing access to too many cmdlets and functions can allow users to do more than you intended and weaken your security stance.

How you go about this process depends on your organization and goals. The following tips can help ensure you're on the right path.

1. **Identify** the commands users are using to get their jobs done. This may involve surveying IT staff, checking automation scripts, or analyzing PowerShell session transcripts and logs.
2. **Update** use of command-line tools to PowerShell equivalents, where possible, for the best auditing and JEA customization experience. External programs can't be constrained as granularly as native PowerShell cmdlets and functions in JEA.
3. **Restrict** the scope of the cmdlets to only allow specific parameters or parameter values. This is especially important if users should manage only part of a system.
4. **Create** custom functions to replace complex commands or commands that are difficult to constrain in JEA. A simple function that wraps a complex command or applies additional validation logic can offer additional control for admins and end-user simplicity.
5. **Test** the scoped list of allowable commands with your users or automation services, and adjust as necessary.

Examples of potentially dangerous commands

Careful selection of commands is important to ensure the JEA endpoint doesn't allow the user to elevate their permissions.

> **Important**
>
> Essential information required for user successCommands in a JEA session are often run with elevated privileges.

The following table contains examples of commands that can be used maliciously if allowed in an unconstrained state. This isn't an exhaustive list and should only be used as a cautionary starting point.

Risk	Example	Related commands
Granting the connecting user admin privileges to bypass JEA	`Add-LocalGroupMember -Member 'CONTOSO\jdoe' -Group 'Administrators'`	`Add-ADGroupMember`, `Add-LocalGroupMember`, `net.exe`, `dsadd.exe`
Running arbitrary code, such as malware, exploits, or custom scripts to bypass protections	`Start-Process -FilePath '\\san\share\malware.exe'`	`Start-Process`, `New-Service`, `Invoke-Item`, `Invoke-WmiMethod`, `Invoke-CimMethod`, `Invoke-Expression`, `Invoke-Command`, `New-ScheduledTask`, `Register-ScheduledJob`

Create a role capability file

You can create a new PowerShell role capability file with the New-PSRoleCapabilityFile cmdlet.

PowerShell Copy

```
New-PSRoleCapabilityFile -Path
.\MyFirstJEARole.psrc
```

The resulting role capability file should be edited to allow the commands required for the role. The PowerShell help documentation contains several examples of how you can configure the file.

Allowing PowerShell cmdlets and functions

To authorize users to run PowerShell cmdlets or functions, add the cmdlet or function name to the VisibleCmdlets or VisibleFunctions fields. If you aren't sure whether a command is a cmdlet or function, you can run `Get-Command <name>` and check the **CommandType** property in the output.

PowerShell Copy

```
VisibleCmdlets = 'Restart-Computer', 'Get-
NetIPAddress'
```

Sometimes the scope of a specific cmdlet or function is too broad for your users' needs. A DNS admin, for example, probably only needs access to restart the DNS service. In multi-tenant environments, tenants have access to self-service management tools. Tenants should be limited to managing their own resources. For these cases, you can restrict which parameters are exposed from the cmdlet or function.

PowerShell

Copy

```
VisibleCmdlets = @{ Name = 'Restart-
Computer'; Parameters = @{ Name = 'Name' }}
```

In more advanced scenarios, you may also need to restrict the values a user may use with these parameters. Role capabilities let you define a set of values or a regular expression pattern that determine what input is allowed.

PowerShell

Copy

```
VisibleCmdlets = @{ Name = 'Restart-Service';
Parameters = @{ Name = 'Name'; ValidateSet =
'Dns', 'Spooler' }},
                  @{ Name = 'Start-Website';
Parameters = @{ Name = 'Name'; ValidatePat-
tern = 'HR_*' }}
```

Note

The **common PowerShell parameters** are always allowed, even if you restrict the available parameters. You should not explicitly list them in the Parameters field.

The table below describes the various ways you can customize a visible cmdlet or function. You can mix and match any of the below in the **VisibleCmdlets**field.

Example	Use case

Example	Use case
`'My-Func'` or `@{ Name = 'My-Func' }`	Allows the user to run `My-Func` without any restrictions on the parameters.
`'MyModule\My-Func'`	Allows the user to run `My-Func` from the module `MyModule` without any restrictions on the parameters.
`'My-*'`	Allows the user to run any cmdlet or function with the verb `My`.
`'*-Func'`	Allows the user to run any cmdlet or function with the noun `Func`.
`@{ Name = 'My-Func'; Parameters = @{ Name = 'Param1'}, @{ Name = 'Param2' }}`	Allows the user to run `My-Func` with the `Param1` and `Param2` parameters. Any value can be supplied to the parameters.

Example	Use case
`@{ Name = 'My-Func'; Parameters = @{ Name = 'Param1'; ValidateSet = 'Value1', 'Value2' }}`	Allows the user to run `My-Func` with the `Param1` parameter. Only "Value1" and "Value2" can be supplied to the parameter.
`@{ Name = 'My-Func'; Parameters = @{ Name = 'Param1'; ValidatePattern = 'contoso.*' }}`	Allows the user to run `My-Func` with the `Param1` parameter. Any value starting with "contoso" can be supplied to the parameter.

Warning

For best security practices, it is not recommended to use wildcards when defining visible cmdlets or functions. Instead, you should explicitly list each trusted command to ensure no other commands that share the same naming scheme are unintentionally authorized.

You can't apply both a **ValidatePattern** and **ValidateSet** to the same cmdlet or function.

If you do, the **ValidatePattern** overrides the **ValidateSet**.

For more information about **ValidatePattern**, check out this *Hey, Scripting Guy!* post and the PowerShell Regular Expressions reference content.

Allowing external commands and PowerShell scripts

To allow users to run executables and PowerShell scripts (.ps1) in a JEA session, you have to add the full path to each program in the **VisibleExternalCommands** field.

PowerShell

Copy

```
VisibleExternalCommands = 'C:\Windows\System32\whoami.exe', 'C:\Program Files\Contoso\Scripts\UpdateITSoftware.ps1'
```

Where possible, to use PowerShell cmdlet or function equivalents for any external executables you authorize since you have control over the parameters allowed with PowerShell cmdlets and functions.

Many executables allow you to read the current state and then change it by providing different parameters.

For example, consider the role of a file server admin that manages network shares hosted on a system. One way of managing shares is to use `net share`. However, allowing **net.exe** is dangerous because the user could use the command to gain admin privileges with `net group Administrators unprivilegedjeauser /add`. A more secure option is to allow Get-SmbShare, which achieves the same result but has a much more limited scope.

When making external commands available to users in a JEA session, always specify the complete path to the executable.

This prevents the execution of similarly named and potentially malicious programs located elsewhere on the system.

Allowing access to PowerShell providers

By default, no PowerShell providers are available in JEA sessions. This reduces the risk of sensitive information and configuration settings being disclosed to the connecting user.

When necessary, you can allow access to the PowerShell providers using the `VisibleProviders` command. For a full list of providers, run `Get-PSProvider`.

PowerShell

Copy

```
VisibleProviders = 'Registry'
```

For simple tasks that require access to the file system, registry, certificate store, or other sensitive providers, consider writing a custom function that works with the provider on the user's behalf. The functions, cmdlets, and external programs available in a JEA session aren't subject to the same constraints as JEA. They can access any provider by default. Also consider using the user drive when copying files to or from a JEA endpoint is required.

Creating custom functions

You can author custom functions in a role capability file to simplify complex tasks for your end users. Custom functions are also useful when you require advanced validation logic for cmdlet parameter values. You can write simple functions in the **FunctionDefinitions** field:

PowerShell

Copy

```
VisibleFunctions = 'Get-TopProcess'

FunctionDefinitions = @{
    Name = 'Get-TopProcess'

    ScriptBlock = {
        param($Count = 10)

        Get-Process | Sort-Object -Property
CPU -Descending |
            Microsoft.PowerShell.Utility\Se-
lect-Object -First $Count
    }
}
```

Important

Don't forget to add the name of your custom functions to the **VisibleFunctions** field so they can be run by the JEA users.

The body (script block) of custom functions runs in the default language mode for the system and isn't subject to JEA's language constraints. This means that functions can access the file system and registry, and run commands that weren't made visible in the role capability file. Take care to avoid running arbitrary code when using parameters. Avoid piping user input directly into cmdlets like `Invoke-Expression`.

In the above example, notice that the fully qualified module name (FQMN)`Microsoft.PowerShell.Utility\Select-Object` was used instead of the shorthand `Select-Object`. Functions defined in role capability files are still subject to the scope of JEA sessions, which includes the proxy functions JEA creates to constrain existing commands.

By default, `Select-Object` is a constrained cmdlet in all JEA sessions that doesn't allow the selection of arbitrary properties on objects. To use the unconstrained `Select-Object` in functions, you must explicitly request the full implementation using the FQMN. Any constrained cmdlet in a JEA session has the same constraints when invoked from a function. For more information, see about_Command_Precedence.

If you're writing several custom functions, it's more convenient to put them in a PowerShell script module. You make those functions visible in the JEA session using the **VisibleFunctions** field like you would with built-in and third-party modules.

For tab completion to work properly in JEA sessions you must include the built-in function `tabexpansion2` in the **VisibleFunctions** list.

Make the role capabilities available to a configuration

Prior to PowerShell 6, for PowerShell to find a role capability file it must be stored in a **RoleCapabilities** folder in a PowerShell module. The module can be stored in any folder included in the `$env:PSModulePath` environment variable, however you shouldn't place it in `$env:SystemRoot\System32` or a folder where untrusted users could modify the files.

The following example creates a PowerShell script module called **ContosoJEA** in the `$env:ProgramFiles` path to host the role capabilities file.

PowerShell

Copy

```
# Create a folder for the module
$modulePath = Join-Path $env:ProgramFiles
```

```
"WindowsPowerShell\Modules\ContosoJEA"
New-Item -ItemType Directory -Path $module-
Path

# Create an empty script module and module
manifest.
# At least one file in the module folder must
have the same name as the folder itself.
New-Item -ItemType File -Path (Join-Path
$modulePath "ContosoJEAFunctions.psm1")
New-ModuleManifest -Path (Join-Path $module-
Path "ContosoJEA.psd1") -RootModule "Contoso-
JEAFunctions.psm1"

# Create the RoleCapabilities folder and copy
in the PSRC file
$rcFolder = Join-Path $modulePath "RoleCapa-
bilities"
New-Item -ItemType Directory $rcFolder
Copy-Item -Path .\MyFirstJEARole.psrc -Desti-
nation $rcFolder
```

For more information about PowerShell modules, see Understanding a PowerShell Module.

Starting in PowerShell 6, the **RoleDefinitions** property was added to the session configuration file. This property lets you specify the location of a role configuration file for your role definition. See the examples in New-PSSessionConfigurationFile.

Updating role capabilities

You can edit a role capability file to update the settings at any time. Any new JEA sessions started after the role capability has been updated will reflect the revised capabilities.

This is why controlling access to the role capabilities folder is so important. Only highly trusted administrators should be allowed to change role capability files. If an untrusted user can change role capability files, they can easily give themselves access to cmdlets that allow them to elevate their privileges.

For administrators looking to lock down access to the role capabilities, ensure Local System has read access to the role capability files and containing modules.

How role capabilities are merged

Users are granted access to all matching role capabilities in the session configuration file when they enter a JEA session. JEA tries to give the user the most permissive set of commands allowed by any of the roles.

VisibleCmdlets and VisibleFunctions

The most complex merge logic affects cmdlets and functions, which can have their parameters and parameter values limited in JEA.

The rules are as follows:

1. If a cmdlet is only made visible in one role, it is visible to the user with any applicable parameter constraints.
2. If a cmdlet is made visible in more than one role, and each role has the same constraints on the cmdlet, the cmdlet is visible to the user with those constraints.
3. If a cmdlet is made visible in more than one role, and each role allows a different set of parameters, the cmdlet and all the parameters defined across every role are visible to the user. If one role doesn't have constraints on the parameters, all parameters are allowed.
4. If one role defines a validate set or validate pattern for a cmdlet parameter, and the other role allows the parameter

but does not constrain the parameter values, the validate set or pattern is ignored.
5. If a validate set is defined for the same cmdlet parameter in more than one role, all values from all validate sets are allowed.
6. If a validate pattern is defined for the same cmdlet parameter in more than one role, any values that match any of the patterns are allowed.
7. If a validate set is defined in one or more roles, and a validate pattern is defined in another role for the same cmdlet parameter, the validate set is ignored and rule (6) applies to the remaining validate patterns.

Below is an example of how roles are merged according to these rules:

PowerShell Copy

```
# Role A Visible Cmdlets
$roleA = @{
    VisibleCmdlets = 'Get-Service',
                     @{ Name = 'Restart-Ser-
vice';
                        Parameters = @{ Name
= 'DisplayName'; ValidateSet = 'DNS Client' }
}
}

# Role B Visible Cmdlets
$roleB = @{
    VisibleCmdlets = @{ Name = 'Get-Service';
                        Parameters = @{ Name
= 'DisplayName'; ValidatePattern = 'DNS.*' }
},
                     @{ Name = 'Restart-Ser-
vice';
                        Parameters = @{ Name
= 'DisplayName'; ValidateSet = 'DNS Server' }
```

```
}
}

# Resulting permissions for a user who be-
longs to both role A and B
# - The constraint in role B for the Display-
Name parameter on Get-Service
#   is ignored because of rule #4
# - The ValidateSets for Restart-Service are
merged because both roles use
#   ValidateSet on the same parameter per
rule #5
$mergedAandB = @{
    VisibleCmdlets = 'Get-Service',
                     @{ Name = 'Restart-Ser-
vice';
                        Parameters = @{ Name
= 'DisplayName'; ValidateSet = 'DNS Client',
'DNS Server' } }
}
```

VisibleExternalCommands, VisibleAliases, VisibleProviders, ScriptsToProcess

All other fields in the role capability file are added to a cumulative set of allowable external commands, aliases, providers, and startup scripts. Any command, alias, provider, or script available in one role capability is available to the JEA user.

Be careful to ensure that the combined set of providers from one role capability and cmdlets/functions/commands from another don't allow users unintentional access to system resources. For example, if one role allows the `Remove-Item` cmdlet and another allows the `FileSystem` provider, you are at risk of a JEA user deleting arbitrary files on your computer. Additional

information about identifying users' effective permissions can be found in the auditing JEA article.