

Windows PowerShell Constrained Endpoints, Proxy Functions, and Just Enough Administration

By Tommy Maynard

PowerShell Pride

On one of the PowerShell-related forums, there was a recently asked question that was rather interesting and thought provoking. While I didn't answer the question on the forum myself, I caught myself pondering the question, and my would-be answer, several times over the following few days. This was the gist of the question:

What have you done with Windows PowerShell of which you're most proud?

There's been a great deal of knowledge absorbed in the last few years by me. It's been an amazing ride. Reading blogs linked from Twitter, turned into linking my own blog on Twitter. Reading questions on PowerShell-related forums, turned into helping with solutions, and writing answers. Being a follower of the PowerShell community transitioned into being an active member. I still desire to encourage those seeking answers from PowerShell, to truly learn PowerShell. The fundamentals are key, and it's easy to spot someone who is trying to learn enough to complete a task, as opposed to someone that has made the decision to provide themselves a true PowerShell education.

What I am most proud of, in relation to my work in PowerShell was this line -- although slightly modified -- that I wrote into a central, change management log in late 2014:

"Removed domain\spadmins from the Local Administrators group on the SharePoint Central Admin Servers."

To ensure you can tell what had been logged, this was the day that I removed the local administrative privileges of an internal team, from our SharePoint Central Admin servers. How did I do that, without removing the ability to complete the internal team's SharePoint responsibilities? It was a Windows PowerShell constrained endpoint. We'll get back to this constrained endpoint shortly, but before we do, let's explore some of the technical aspects of creating a constrained endpoint.

Building a Constrained Endpoint

The terms endpoint, constrained endpoint, and session configuration are used somewhat interchangeably. This is true for proxy function and custom function, too. A session configuration, or endpoint, is a remote set of configurations that define the user experience when using a PowerShell remote session. A constrained endpoint is a session configuration, or endpoint, that has been configured to only allow a specific set of functionalities, thus limiting its purpose, and creating a security boundary through the limitation of cmdlets and functions. Constrained endpoints use an elevated, RunAsUser account. This allows non-administrative users, once given the proper permissions to the endpoint, the ability to perform administrative tasks without the need to ever have their account elevated.

I'm not out to provide *all* the possible details and possibilities when you create a constrained endpoint; however, I am going to walk through enough to get you started creating your own. That said, be sure to read the JEA (Just Enough Administration) portion at the end of this article. JEA is the highly preferred way to create constrained endpoints -- JEA endpoints -- at scale. In addition, they offer several benefits over their constrained endpoint counterparts. I believe that a solid understanding of PowerShell constrained endpoints, will help anyone better understand JEA. My own introduction to JEA was quite pleasant, based on the knowledge and experience that I already had.

The first recommendation is that the system where an endpoint is created and deployed be running a minimum of Windows Management Framework 3 (WMF) -- that package includes PowerShell 3.0. This is the default version on Windows Server 2012 and Windows 8. Included in PowerShell 3.0 was a helpful way to create session configurations: the `New-PSSessionConfigurationFile` cmdlet. You need to be certain that PowerShell Remoting is available and functioning. If it's not enabled, read about the `Enable-PSRemoting` cmdlet. You'll know if it is, or not, when you run the example commands below (replace `SPCA01` with the name of your server). Start by entering `Get-PSSessionConfiguration` on the system in which you want to connect. This cmdlet will indicate the session configurations, or endpoints, available on the server. When the `Get-PSSessionConfiguration` cmdlet is run, you'll likely see a few different session configurations, one of which is named `Microsoft.PowerShell`. This is the default endpoint. That's to say, that if you don't enter the `-ConfigurationName` parameter, and provide a value for it, you will connect to this endpoint, if you have the proper permissions. By default, you must be a local administrator to use the default endpoint. Take a look at the two examples

using Enter-PSSession and Invoke-Command below. Both of these are connecting to the Microsoft.PowerShell default endpoint on the SPCA01 server.

```
PS C:\> Enter-PSSession -ComputerName SPCA01
[SPCA01]: PS C:\Users\tommymaynard\Documents> Get-PSSessionConfiguration | Select-Object Name

Name
----
Microsoft.PowerShell
microsoft.powershell.workflow
microsoft.powershell32
microsoft.windows.servermanagerworkflows

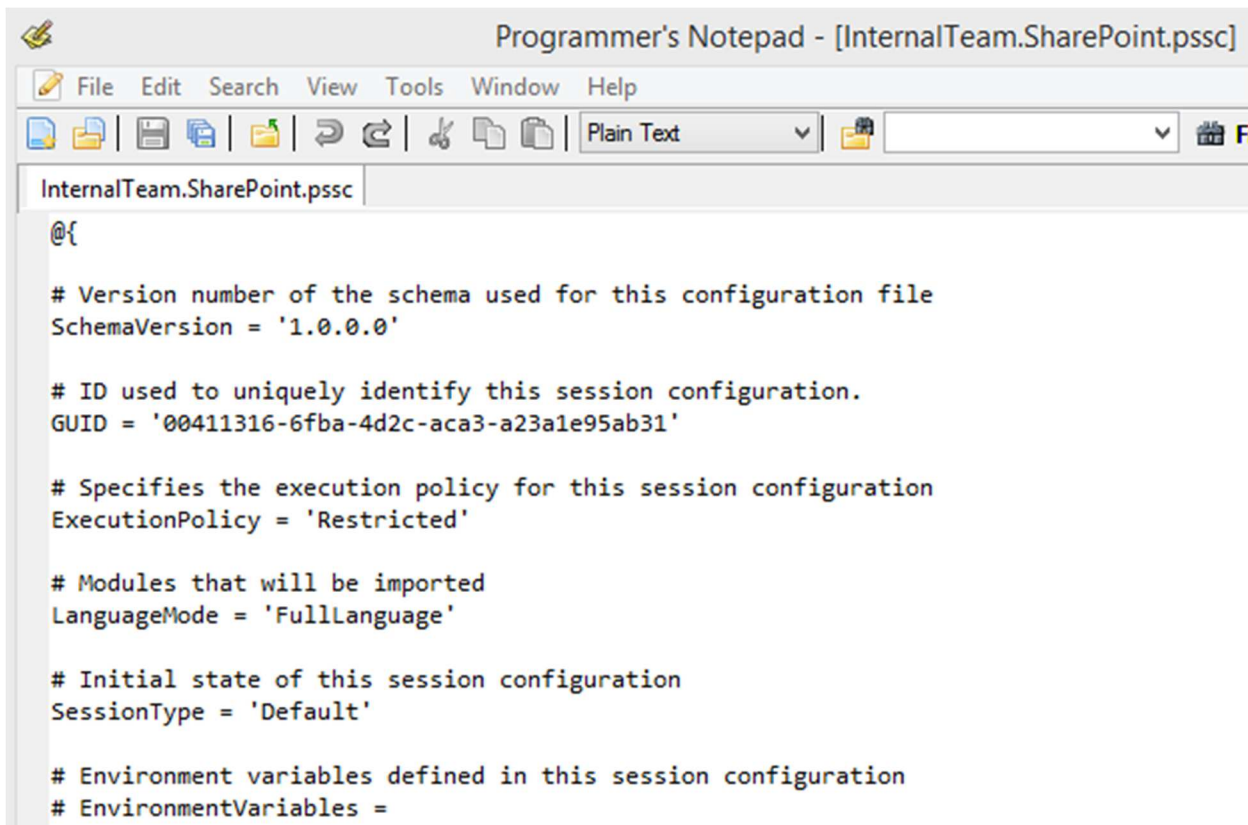
[SPCA01]: PS C:\Users\tommymaynard\Documents> Exit-PSSession

PS C:\> Invoke-Command -ComputerName SPCA01 -ScriptBlock {Get-PSSessionConfiguration | Select-Object Name}

Name
----
Microsoft.PowerShell
microsoft.powershell.workflow
microsoft.powershell32
microsoft.windows.servermanagerworkflows
```

As previously mentioned, we can create session configuration files using the New-PSSessionConfigurationFile cmdlet. The only required parameter is the -Path parameter. The value provided to this parameter is the location where the session configuration file is saved. It will need to include the file name and a .pssc file extension. If you don't include the proper file extension, the cmdlet will show an error. Even though all we have to use is -Path, there are several parameters for which we can provide values. It's basically one for each setting inside the file. On that note, if you haven't created a session configuration file, you might consider doing that and opening it in a text editor. This file defines everything about the constrained endpoint. It includes the mundane: the Author, the Copyright, and the CompanyName, to the more advanced: ModulesToImport, VisibleCmdlets, VisibleFunctions, and ScriptsToProcess. The example below creates a session configuration file and then opens it. I've included a partial look at the file it created, just below this example.

```
PS C:\> New-PSSessionConfigurationFile -Path 'C:\ConstrainedEPs\InternalTeam.Sharepoint.pssc'
PS C:\> Invoke-Item -Path 'C:\ConstrainedEPs\InternalTeam.Sharepoint.pssc'
```



```
@{  
  
# Version number of the schema used for this configuration file  
SchemaVersion = '1.0.0.0'  
  
# ID used to uniquely identify this session configuration.  
GUID = '00411316-6fba-4d2c-aca3-a23a1e95ab31'  
  
# Specifies the execution policy for this session configuration  
ExecutionPolicy = 'Restricted'  
  
# Modules that will be imported  
LanguageMode = 'FullLanguage'  
  
# Initial state of this session configuration  
SessionType = 'Default'  
  
# Environment variables defined in this session configuration  
# EnvironmentVariables =
```

If you don't uncomment `VisibleCmdlets` from inside the newly created session configuration file, and enter a list of cmdlets, then the endpoint will not offer any cmdlet filtering. All of the cmdlets will be available to the endpoint's user. Once you enter a cmdlet for this property, it will begin restricting what the user will see when connected to the endpoint. I would recommend you add the two lines below. Then, add additional cmdlets and functions you want visible in your endpoint, as well. The cmdlets and functions below are there to create a pleasant experience when connected to the constrained endpoint. Without many of these cmdlets your users will end up seeing some errors, too. For instance, `Get-Command` is run behind the scenes when you connect to an endpoint -- something you'll quickly find out if `Get-Command` isn't listed as one of the visible cmdlets.

Cmdlets visible in this session configuration

```
VisibleCmdlets = 'Exit-PSSession','Format-List','Format-Table','Get-Command','Get-FormatData','Get-Help','Measure-Object','Out-Default','Select-Object'
```

Functions visible in this session configuration

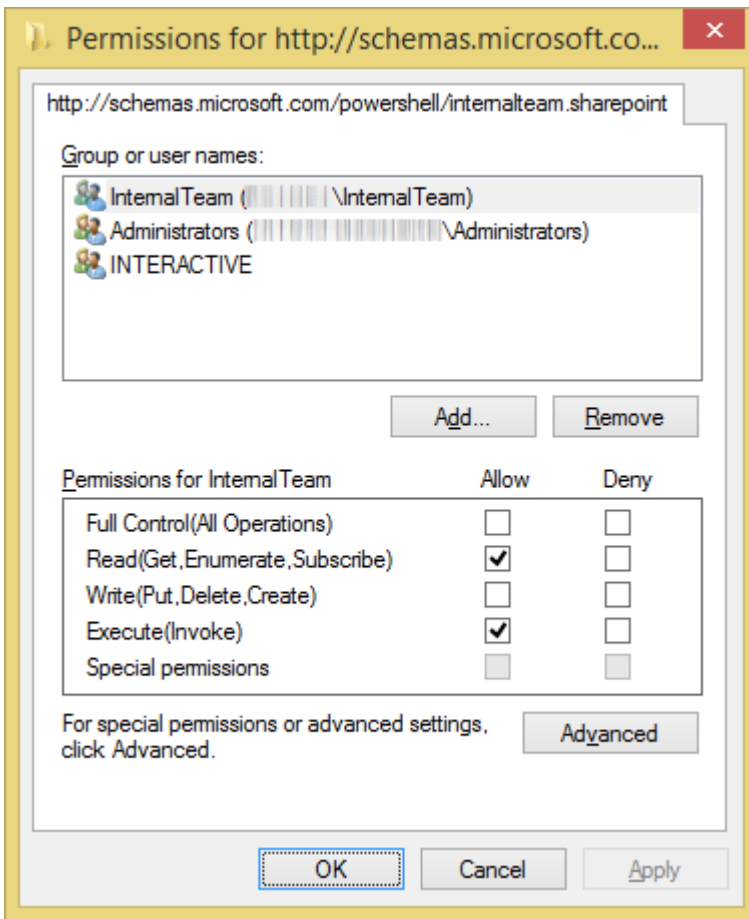
```
VisibleFunctions = 'prompt','TabExpansion2'
```

Once you've modified this file to the desired preference, close it and save it. This is, of course, if you didn't use the parameters included with the `New-PSSessionConfigurationFile` cmdlet, to enter the values when the file was originally created. When you feel like the session configuration is ready for testing, you need to register it on the server, but before doing that, try the `Test-PSSessionConfiguration` cmdlet. Its purpose is to ensure a session configuration file contains valid keys and values. If it returns 'False,' rerun the command and include the `-Verbose` parameter. This will help identify where the cmdlet has found improper keys and values. I've seen this cmdlet fail on pinpointing the exact problem, but it will usually get you close.

When an endpoint is registered using the `Register-PSSessionConfiguration` cmdlet, there are several things that happen.

```
PS C:\> Register-PSSessionConfiguration -Name InternalTeam.SharePoint -Path  
'C:\ConstrainedEPs\InternalTeam.Sharepoint.pssc' -RunAsCredential SPCA01\spadmin  
-ShowSecurityDescriptorUI
```

First, you'll get a prompt to provide the password for the account that will be used to run the endpoint. In my example, I've used the local administrator account SPCA01\spadmin. It's always best to use a local administrative account to run a constrained endpoint. This will prevent an attacker from moving horizontally -- compromising one server after another -- if the account becomes compromised. Next, you'll get a warning. It's important to understand that constrained endpoints, when not fully thought out, may allow an endpoint user to do things you didn't intend, or consider. After the warning is a prompt to verify that you really want to create the endpoint, and another to approve the restart of the WinRM service -- a requirement to make the constrained endpoint usable. At the near end of this process, you'll be presented with a security dialog, as in the image below. It's this dialog where you add the group -- often an Active Directory (AD) group -- that will have permission to use this endpoint. This is a group of non-administrators that we will allow to run as the SPCA01\spadmin local administrator account. Choose the options for Read and Execute. In the image, the <DOMAIN>\InternalTeam AD group has been added, and the proper permissions have been applied.



When this process is completed, it will have created a session configuration file located at C:\Windows\System32\WindowsPowerShell\v1.0\SessionConfig. The file is named after the value supplied to the -Name parameter in the Register-PSSessionConfiguration cmdlet, followed by an underscore and the GUID from inside the session configuration file itself. This GUID was automatically generated when the file was first created by the New-PSSessionConfigurationFile cmdlet.

If you want to make changes to the session configuration file, I recommend that you unregister the endpoint and edit your original file (at C:\ConstrainedEPs) -- adding or removing a cmdlet, for instance -- and then reregistering the session configuration. You can edit the file in the SessionConfig directory directly. Even so, I've only ever done it that way to test if that works, or not. I always unregister and reregister my constrained endpoints, as a way to ensure my original file matches the session configuration file in the SessionConfig directory.

Now that your constrained endpoint has been registered, let's connect and run a command, Get-Command, inside the session. While you can connect interactively, using Enter-PSSession, you can also use Invoke-Command. We've already seen both of these when we connected to SPCA01 server with the default, Microsoft.PowerShell endpoint. About my

naming convention: It is not required to use a dot in the endpoint's name (Internal.SharePoint). I've chosen to do this based on how Microsoft has named their default endpoints. It has allowed for a logical naming convention, where I separate the name of the team that will use the endpoint (using a dot), from the endpoint's purpose.

```
PS C:\> Invoke-Command -ComputerName SPCA01 -ConfigurationName InternalTeam.SharePoint -ScriptBlock
{Get-Command | Format-Table}
```

CommandType	Name	ModuleName
Function	prompt	
Function	TabExpansion2	
Cmdlet	Exit-PSSession	Microsoft.PowerShell.Core
Cmdlet	Format-List	Microsoft.PowerShell.Utility
Cmdlet	Format-Table	Microsoft.PowerShell.Utility
Cmdlet	Get-Command	Microsoft.PowerShell.Core
Cmdlet	Get-FormatData	Microsoft.PowerShell.Utility
Cmdlet	Get-Help	Microsoft.PowerShell.Core
Cmdlet	Get-Process	Microsoft.PowerShell.Management
Cmdlet	Get-Service	Microsoft.PowerShell.Management
Cmdlet	Measure-Object	Microsoft.PowerShell.Utility
Cmdlet	Out-Default	Microsoft.PowerShell.Core
Cmdlet	Select-Object	Microsoft.PowerShell.Utility

You can unregister an endpoint using the Unregister-PSSessionConfiguration cmdlet, as I've done below. The -Name parameter takes wildcards, so be careful if you use them and have similarly named session configurations. Much like when you register an endpoint, you'll need to confirm you really want to remove the endpoint, and that it's a suitable time to restart the WinRM service. Restarting this service while someone is actively using a PS Remoting session, will break the session immediately. This holds true for when the endpoint is first registered, as well, as the WinRM service is required to restart then, too.

```
PS C:\> Unregister-PSSessionConfiguration -Name Internal*
```

Although there is more one can do with session configurations and the discussed cmdlets, this is a fair introduction and a primer to JEA.

The Internal Team SharePoint Constrained Endpoint

Before my SharePoint constrained endpoint, we allowed a small group of people, which were not in our Infrastructure team, the ability to run as local administrators on the SharePoint Central Admin servers. Frightening, and as it turns out, totally unnecessary.

This group had the ability to use Remote Desktop (because they were local administrators), to connect to the servers. Once they had established a remote desktop session, they would open the SharePoint 2013 Management Shell to add new SharePoint Site Collections. While it was mildly nerve-wracking to remove their local administrative access that day -- for the fear of breaking something -- it was also rewarding, and eventually my proudest moment in PowerShell. It was also the beginning of other constrained endpoints and proxy functions. Next thing you know, I've delegated work back to those that had requested it; I had removed my team from other team's workflows, and collectively saved time for everyone involved. One developer, using a different constrained endpoint, later said to me, "This is how I wanted it from the beginning."

Once the constrained endpoint was in place, our internal team connected via their PowerShell consoles, and our PowerShell Web Access server, to create new SharePoint Site Collections. It was a couple months in, when they requested the Set-SPSite cmdlet to edit existing Site Collections. They already had access to a few Get-SharePoint cmdlets, and the New-SPSite SharePoint cmdlet, to create new Site Collections. We discussed the request among the infrastructure team and decided we would allow use of the Set-SPSite cmdlet. I unregistered the endpoint, added the cmdlet to the list of visible cmdlets, and registered the endpoint.

Another couple months passed, and another request came in, but this time for the Remove-SPSite cmdlet. All the individuals on the internal team are people we trusted, but giving them this level of destruction -- we'll call it -- was just too much responsibility to hand over. Our denial did, however, come with a possibility: a proxy function. I wrote the

internal team, “We have decided not to provide the full, Remove-SPSite cmdlet. I have, however, written a custom function that should provide you the ability to delete Site Collections (SCs), while maintaining some precautions.” I went on to say, “This function will permit you to remove SharePoint SCs under the condition they were created within the last six hours. This seems like a reasonable compromise since some of your SCs often need to be deleted close in time to their creation.”

They said they’d let me know if six hours wasn’t enough time, and I never heard from them again. My team never removed another Site Collection for them again, and we never lost an important, longstanding Site Collection, as well. We couldn’t have; we had protected ourselves against that risk.

One of the things to fully digest here, is that the proxy function below allowed for the removal of a Site Collection using the Remove-SPSite cmdlet, when that internal team didn’t have access to use that cmdlet. That’s one of the beauties of a proxy function: It can internally use cmdlets that those invoking the function don’t have direct access to use.

```
Function Remove-SPSite {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory = $true, Position = 0, ValueFromPipeline = $True)]
        [Microsoft.SharePoint.PowerShell.SPSitePipeBind]$Identity
    )
    Begin {} # End Begin.
    Process {
        ##### 1st.
        If (Get-PSSnapin -Name Microsoft.SharePoint.PowerShell -ErrorAction SilentlyContinue) {
            $SPSite = Get-SPSite -Identity $Identity -ErrorAction SilentlyContinue

            ##### 2nd.
            If ($SPSite) {
                $Hours = 6
                $TimeHoursAgo = ((Get-Date).ToUniversalTime()).AddHours(-$Hours)

                ##### 3rd.
                If ($SPSite.RootWeb.Created -gt $TimeHoursAgo) {
                    Microsoft.SharePoint.PowerShell\Remove-SPSite `
                        -Identity $SPSite.Url `
                        -Confirm:$True
                    $ReCheck = Get-SPSite -Identity $Identity -ErrorAction SilentlyContinue

                    ##### 4th.
                    If ($ReCheck) {
                        Write-Output -InputObject `
                            "Site Collection was not deleted: $($SPSite.Url)."
                    } Else {# 4th.
                        Write-Output -InputObject "Site Collection was deleted: $($SPSite.Url)."
                    }

                } Else {# 3rd.
                    Write-Warning -Message `
                        "Unable to delete Site Collection: $($SPSite.Url) [Over $Hours hours old]."
                }

            } Else {# 2nd.
                Write-Warning -Message `
                    "Unable to locate Site Collection: $($Identity.SiteUrl)."
            }

        } Else {# 1st.
            Write-Warning -Message `
                'Cannot detect that the SharePoint PowerShell PSSnapin has been loaded.'
        }
    } # End Process.
    End {} # End End.
} # End Function: Remove-SPSite.
```

Here are some key points about this function. First, this proxy function is going to require that the SharePoint PSSnapin is added *before* this function is *ever* run. This is because the -Identity parameter, set in the Param() block, is set to accept a specific SharePoint data type: Microsoft.SharePoint.PowerShell.SPSiteBind. We’re doing this in order to make this

function act as much like the Remove-SPSite cmdlet as possible. This means we can't add the PSSnapin inside this function. Therefore, the SharePoint PSSnapin is added as a part of a ScriptsToProcess file set in our .pssc file as `ScriptsToProcess = 'C:\ConstrainedEPs\InternalTeam.SharePoint.ScriptsToProcess.ps1'`. The ScriptsToProcess.ps1 file is loaded just before the user connects to this endpoint, ensuring the PSSnapin is added. It has one simple line: `Add-PSSnapin -Name 'Microsoft.SharePoint.PowerShell'`. Even though we've instructed the endpoint to add the PSSnapin, we still check for it in this function.

Second, in the third If statement we invoke the Remove-SPSite SharePoint cmdlet. Before we do that though, this is where we compare the current time (once adjusted to UTC time) and the time in which the Site Collection was created. To get inside this If statement, the Site Collection will have had to been created in the last six hours. Once that condition is satisfied, notice that I used the full path to the SharePoint cmdlet as `Microsoft.SharePoint.PowerShell\Remove-SPSite`. Had I not included the cmdlet this way, PowerShell would have thought I was trying to invoke this same function again. This function was named Remove-SPSite, creating a naming overlap. Keep in mind the command precedence rules. Functions are run before cmdlets, if there's a function and cmdlet with the same name (see `about_Command_Precedence`).

Providing all this passes, we enter the statement and remove the Site Collection. I opted to recheck if the Site Collection was actually deleted or not. That's the purpose behind the final, nested If Statement that sets the \$ReCheck variable. Before we move on to the next constrained endpoint, let's remind ourselves of something. We just allowed this function to use a cmdlet that a user who invoked the function, couldn't have run otherwise. The function is not limited by the visible cmdlets of the constrained endpoint.

The Third-Party Application Constrained Endpoint

A few times per week, the developers that were supporting a third-party application often needed our team to perform the same task for them. While I wasn't the team member that supported this project directly, I was tired of seeing this repetitive request. It went like this: (1) replace two files on server A, B, or C (think Dev, Test, Prod), (2) delete a folder from the same server, and (3) restart two services, also on the same server. With the combination of DFS and shadow copies, I was able to open up these servers so the developers were able to replace the files themselves, and allow for file restores, if necessary: (1) and (2) were solved. With a constrained endpoint, I was able to solve the last requirement.

I created a single, two function module. One function was called Get-Service. Its purpose was to allow the developers the ability to see the status of *only* two services. Here's that function, without the comment-based help and verbose statements. It's real simple: Only inform the user of the status of the Apache2.2 and Tomcat6 services (yes, this is Apache on Windows). There was no reason the developers needed to know anything about any of the other services running on these servers, therefore this proxy function was written, in order to hide the default output of the Get-Service cmdlet. Notice again, we used the full path to the Get-Service cmdlet since the function is named Get-Service.

```
Function Get-Service {
    [CmdletBinding()]
    Param ()

    Begin {} # End Begin.

    Process {
        try {
            Microsoft.PowerShell.Management\Get-Service `
                -Name 'Apache2.2', 'Tomcat6'
                -ErrorAction Stop
        } catch [Microsoft.PowerShell.Commands.ServiceCommandException] {
            Write-Warning -Message 'Cannot find service(s).'
        }
    } # End Process.

    End {} # End End.
} # End Function: Get-Service.
```

Both comment-based help and verbose statements are going to be crucial when you hand your tools over to an individual, or group. This is especially true for those people and groups you encourage to utilize PowerShell, as opposed to those that already have an interest and background.

The purpose of the Edit-Service function was to allow the same developers the ability to stop, start, or restart either of the same two services. If I had to guess, I suspect that the two files they added to the file system, would recreate the folder they had previously deleted, when the service(s) was restarted. The function was written with two parameters. The first one is -Service where they can enter the name, or names, of the service where they want to apply an action. It only accepts either Apache2.2 or Tomcat6, or both values, separated by a comma. The second parameter is -Action and it only accepts one value: Stop, Start, or Restart. When run, it takes the chosen action against whatever service(s) is included. Let's take a look.

```

Function Edit-Service {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory = $true, Position = 0)]
        [ValidateSet('Apache2.2', 'Tomcat6')]
        [string[]]$Service,

        [Parameter(Mandatory = $true, Position = 1)]
        [ValidateSet('Start', 'Stop', 'Restart')]
        [string]$Action
    )

    Begin {} # End Begin.

    Process {
        Foreach ($SingleService in $Service) {
            try {
                $ServiceStatus = (Microsoft.PowerShell.Management\Get-Service `
                    -Name $SingleService `
                    -ErrorAction Stop).Status
                $Proceed = $true
            } catch {
                Write-Warning -Message "Cannot find service ($SingleService)."
                $Proceed = $false
            }

            If ($Proceed -eq $true) {
                Switch ($Action) {
                    'Start' {
                        try {
                            If ($ServiceStatus -ne 'Running') {
                                Microsoft.PowerShell.Management\Start-Service `
                                    -Name $SingleService -Verbose
                            } Else {
                                Write-Warning -Message `
                                    "Service is already running ($SingleService)."
                            }
                        } catch [Microsoft.PowerShell.Commands.ServiceCommandException] {
                            Write-Warning -Message "Cannot find service ($SingleService)."
                        } catch {
                            Write-Warning -Message "Cannot start service ($SingleService)."
                        }
                    } # End Start.

                    'Stop' {
                        try {
                            If ($ServiceStatus -ne 'Stopped') {
                                Microsoft.PowerShell.Management\Stop-Service `
                                    -Name $SingleService -Verbose
                            } Else {
                                Write-Warning -Message `
                                    "Service is already stopped ($SingleService)."
                            }
                        } catch [Microsoft.PowerShell.Commands.ServiceCommandException] {
                            Write-Warning -Message "Cannot find service ($SingleService)."
                        } catch {
                            Write-Warning -Message "Cannot stop service ($SingleService)."
                        }
                    } # End Stop.

                    'Restart' {
                        try {
                            If ($ServiceStatus -ne 'Stopped') {
                                Microsoft.PowerShell.Management\Restart-Service `
                                    -Name $SingleService -Verbose
                            } Else {
                                Write-Warning -Message "Service is stopped ($SingleService)."
                            }
                        } catch [Microsoft.PowerShell.Commands.ServiceCommandException] {
                            Write-Warning -Message "Cannot find service ($SingleService)."
                        } catch {
                            Write-Warning -Message "Cannot restart service ($SingleService)."
                        }
                    } # End Restart.
                } # End Switch.
            } # End If.
        } # End Foreach.
    } # End Process.

    End {} # End End.
} # End Function: Edit-Service.

```

It's a busy function, and could likely use some writing improvements. Even so, this function allows the developers the ability to change the status of these two services quite well. The only thing it seems to be missing is some logging. The function could have collected the currently connected user (from `$PSSenderInfo.ConnectedUser`), and dropped that into a text file, to include the services and action they took with each run of the function. There hasn't been a need for this logging, but for the sake of being as complete as possible, this may have been a nice feature to add. Even better, I could have included not only the logging, but a third custom function whose sole purpose would be to read this log file. This would allow the developers to see the last time these services were stopped, started, or restarted by one of them, and which of them initiated the action(s). All this, without knowing how to use PowerShell to return the contents of a file, or having access to the `Get-Content` cmdlet. This is another great reason for proxy functions: we can hide, or wrap, cmdlets we aren't expecting our users to know, or know how to use.

The Video Team Constrained Endpoint

My final constrained endpoint was for a video team and their small dependence on Active Directory. Put another way: They had a dependence on *me* and my ability to modify things in Active Directory *for them*. Although I can modify group memberships and user accounts in AD rather quickly by using PowerShell, I had finally decided I would give this work back to them, by using another constrained endpoint and five proxy functions. Since this constrained endpoint was deployed, our team hasn't once needed to make any AD modifications for this team. There's a theme here: securely speed up the workflow, and decrease the dependence on our team.

There were two base things the video team needed to accomplish: add, remove, and view the membership of one AD group, and expire and unexpire one AD user when necessary. We'll start with the functions written for the AD group first, leaving in the `Write-Verbose` statements. If you're going to write functions for non-PowerShell users, then make sure to use complete comment-based help, and liberal and detailed use of `Write-Verbose`. It's important to understand that you're writing these tools for others, not yourself. These `Write-Verbose` commands help inform the user what happens, as the function runs. This first function, `Get-ADGroupMember`, returns the current members of `$ADGroup`. As you'll soon see, `$ADGroup` isn't defined inside the function. Because these functions are a part of a module, I'm able to define the variable at the module-level (inside the `.psm1` file, and outside of any of the contained functions), thus making the variable useable to all the functions.

The first of the three functions that dealt with the AD group members will simply retrieve all the members of the group stored in `$ADGroup`. While the goal was to add and remove users from their group, I wanted to provide the endpoint users a way to see who was currently a member, as well as a way to verify if a user had been successfully added or removed.

```
Function Get-ADGroupMember {
    [CmdletBinding()]
    Param ()

    Begin {
        Write-Verbose -Message "Beginning function: $($MyInvocation.InvocationName)"
    } # End Begin.

    Process {
        try {
            Write-Verbose -Message "Attempting to return users in $ADGroup."
            ActiveDirectory\Get-ADGroupMember -Identity $ADGroup |
                Select-Object @{N='User';E={$_.SamAccountName}},Name |
                Tee-Object -Variable CountOf |
                Sort-Object User
            Write-Verbose -Message "Current number of members: $($CountOf.Count)."
        } catch {
            Write-Warning -Message 'Unable to query Active Directory.'
        } # End try-catch.
    } # End Process.

    End {
        Write-Verbose -Message "Ending function: $($MyInvocation.InvocationName)"
    } # End End.
} # End Function: Get-ADGroupMember.
```

The second AD group-related function is Add-ADGroupMember. As the name suggests, this custom function allowed the endpoint's users the ability to add users to the group stored in \$ADGroup. I've borrowed my function names from existing AD cmdlet names. This means we've had to use the full path to the actual cmdlet, such as *ActiveDirectory\Add-ADGroupMember*.

```
Function Add-ADGroupMember {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory = $true)]
        [string[]]$User
    )

    Begin {
        Write-Verbose -Message "Beginning function: $($MyInvocation.InvocationName)"
    } # End Begin.

    Process {
        Foreach ($U in $User) {
            try {
                Write-Verbose -Message "Attempting to add user(s) to $ADGroup."
                ActiveDirectory\Add-ADGroupMember -Identity $ADGroup -Members $U -ErrorAction Stop
            } catch [Microsoft.ActiveDirectory.Management.ADIdentityNotFoundException] {
                Write-Warning -Message "Unable to locate user in Active Directory (User: $U)."
            } catch {
                Write-Warning -Message 'Unknown Error.'
            }
        } # End Foreach.
    } # End Process.

    End {
        Write-Verbose -Message "Ending function: $($MyInvocation.InvocationName)"
    } # End End.
} # End Function: Add-ADGroupMember.
```

The final function of these three, Remove-ADGroupMember, which I've opted not to include, only has two differences from the Add-ADGroupMember function above. One, the name, and two, the line that calls the AD cmdlet. In the remove function, I've replaced *ActiveDirectory\Add-ADGroupMember* with the opposite cmdlet, *ActiveDirectory\Remove-ADGroupMember*.

One of the last two proxy functions for this endpoint, allows the users the ability to determine if a single, specific AD user account is expired, or not. The final proxy function will allow them to expire and unexpired the same user account. This AD user account has been used for video interviews and will often need to be briefly unexpired, used as part of a remote interview, and then expired. The first function is straightforward, returning the Name, SamAccountName, and the user account's current expiration date. If the returned date has already passed, then the account is expired and is not useable for an interview. Much like the \$ADGroup variable, the \$ADUser variable has been defined at the module-level, so it can be defined once, and then used in multiple functions.

```
Function Get-ADAccountExpiration {
    [CmdletBinding()]
    Param ()

    Begin {} # End Begin.

    Process {
        try {
            ActiveDirectory\Get-ADUser $ADUser -Properties AccountExpirationDate |
            Select-Object Name, SamAccountName, AccountExpirationDate
        } catch {
            Write-Warning -Message "Unable to locate account."
        }
    } # End Process.

    End {} # End End.
} # End Function: Get-ADAccountExpiration.
```

In order to make the account useable for a video interview, it needs to be unexpired using the Set-ADAccountExpiration custom function. Once it's no longer needed, it needs to be expired. Both of these options are available using the same

function. Much like the Edit-Service function used by our third-party software developers, I've included an -Action parameter in this function. It'll accept either the value of Expire, or Unexpire.

```
Function SetADAccountExpiration {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory = $true)]
        [ValidateSet('Expire','Unexpire')]
        [string]$Action
    )

    Begin {} # End Begin.

    Process {
        If ($Action -eq 'Unexpire') {
            try {
                ActiveDirectory\Set-ADAccountExpiration `
                    -Identity $ADUser `
                    -DateTime ((Get-Date).AddDays(14))
                Get-ADUser $ADUser -Properties AccountExpirationDate |
                Select-Object Name,SamAccountName,AccountExpirationDate
            } catch {
                Write-Warning -Message "Unable to set account Expiration Date."
            }
        } ElseIf ($Action -eq 'Expire') {
            try {
                ActiveDirectory\Set-ADAccountExpiration `
                    -Identity $ADUser `
                    -DateTime ((Get-Date).AddDays(-14))
                Get-ADUser $ADUser -Properties AccountExpirationDate |
                Select-Object Name,SamAccountName,AccountExpirationDate
            } catch {
                Write-Warning -Message "Unable to set account Expiration Date."
            }
        }
    } # End Process.

    End {} # End End.
} # End Function: Set-ADAccountExpiration.
```

With these five functions, our video team can fully manage the pieces of AD in which their software and appliances are dependent. Again, the combination of the PowerShell constrained endpoint and a few small functions allowed us to speed up the workflow of one team, and remove something from the workflow of another (mine). In all but the first constrained endpoint for SharePoint, we only allow our endpoint users to use the endpoint via our PowerShell Web Access Server. This required one small firewall exception to the PowerShell Web Access Server in the endpoint users' respective VPN profiles.

Why JEA Makes Sense

One of the things I did recently was read the JEA whitepaper from front to back. JEA, which I mentioned earlier, stands for Just Enough Administration. This concept is quite close in relation with constrained endpoints with some distinct differences.

One difference, is that we use Desired State Configuration (DSC) as the method used to create and distribute configuration documents (MOF files), to the server where we want to have a new endpoint(s). These MOF files are documents that define how the server should configure itself, so that it has a JEA endpoint, and a JEA toolkit. More about the JEA toolkit in a moment.

Providing you understand DSC, then this will remove some of the complexity of constrained endpoints, especially when used with a greater number of servers. My three constrained endpoints were added to five different servers, three of which were Dev, Test and Prod for our third-party application software. The other two, were actually on the same utility server. While I would have likely been able to script the delivery of my constrained endpoints, DSC seems like a much better option to deliver and ensure that my endpoints are created, updated when changes are made to their configurations, and put back in working order ("in state"), if someone were to remove them.

Another difference is the built-in logging that takes place with JEA. When someone connects to a standard constrained endpoint, the logging is quite minimal. With JEA, we get logging for all the operations that are performed. This includes the process ID, the RunspaceID, information about WinRM, and all commands run in the JEA session. These are all written to the Windows Event Logs. Due to the fact the endpoint is operating as the JEA account, there's a CSV document we can use to match the user that connected to the JEA endpoint and the RunspaceID, to determine the individual that connected to the JEA endpoint.

Proxy functions are used less since JEA uses a JEA toolkit. This is the explanation, or definition, of what cmdlets and functions can be run in the endpoint. While we can determine the cmdlets and function allowed with constrained endpoints, JEA takes this further. JEA will allow us to choose which parameter(s) can be use with the allowed cmdlets and functions. In addition, it will also let us determine the values that can be provided to the allowed parameters. We can define what patterns can be used for the parameter values, as well. If you think back to the third-party application endpoint, I would have saved any time lost developing the Get-Service and Edit-Service functions. Instead, I could have defined a toolkit that allowed the use of the actual Get-Service cmdlet with the -Name parameter that allowed for either, or both, of the values Apache2.2 and Tomcat6. In addition, I could have never written Edit-Service and instead allowed the use of Stop-Service, Start-Service, and Restart-Service with the same accepted parameter and values. There's a clear savings here in time and complexity. Someone that doesn't have the ability to write a proxy or custom function, can still build a JEA toolkit.

Lastly, JEA handles all the RunAsUser accounts. For every JEA endpoint, it will create a local account with a strong, unknown password, under which the endpoint will run. While domain accounts can be used as the RunAsUser account for a constrained endpoint, JEA uses local accounts in case the JEA endpoint account is ever compromised. If this were to happen, the account would not be useable to move horizontally through an organization. If the endpoint user, as opposed to the RunAsUser, ever need to access remote resources, then they would need to use their own credentials to connect.

We've covered what a constrained endpoint is, how we can create them, and how we can use our own custom functions to assist our endpoint users. Consider much of the article a preparation document for better understanding JEA. It's the better option where we're running PowerShell 4.0 and 5.0, the two versions thus far, that included Desired State Configuration.

Final Note

At nearly the same time this article was written, a new whitepaper was released entitled, *Just Enough Administration (JEA) Infrastructure: An Introduction*, by John Slack. This covers many of the changes in the non-experimental version of JEA included in WMF 5 (PowerShell 5.0). The experimental version, xJEA, was released after PowerShell 4.0 was released, and has been the version discussed in the brief JEA portion of this article. I've only skimmed over the document and I can *already* see some improvements. For instance, the local administrator (JEA) accounts used by each JEA endpoint will only exist when the endpoint is being used (the JEA account is now being called a Virtual Account). It's created when the remote session starts, and destroyed when the remote session ends. This means that the JEA local administrator account is never sitting on the server unless someone is *actively* connected to the endpoint. It's fair to believe that there's more to come in Windows PowerShell and Just Enough Administration. In closing, please continue to remember and understand that administrators are an attack surface. I think it's fair to believe that JEA, and constrained delegation, will continue to grow past what is already being reimagined, with even the newest release of Just Enough Administration.