

# TEC US 2011: PowerShell Deep Dive

## Aleksandar Nikolić

### Constrained PowerShell Endpoints

---

These are the scripts Alexandar was using. You can also find his:

- Slides here: <http://dmitrysootnikov.wordpress.com/2011/08/18/slides-from-deep-dive-sessions/>
- Video recording here: <http://dmitrysootnikov.wordpress.com/2011/08/08/deep-dive-video-constrained-powershell-endpoints-aleksandar-nikolic/>

#### TestModule.psm1

```
Function Get-PrivateFunction {
Write-Host "This is a private function."
}

Function Get-PublicFunction {
Write-Host "This is a public function."
}

Export-ModuleMember -Function Get-PublicFunction
```

#### constrained.ps1

```
# phase 1

# hide all existing commands
foreach ($cmd in Get-Command)
{
    $cmd.Visibility = "private"
}

# phase 2 (remove block comment)
<#

# hide all variables
foreach ($var in Get-Variable)
{
    $var.Visibility = "private"
}

# hide all applications and scripts
$ExecutionContext.SessionState.Applications.Clear()
$ExecutionContext.SessionState.Scripts.Clear()

# set NoLanguage mode
$ExecutionContext.SessionState.LanguageMode = "NoLanguage"
```

```

#>
# end of phase 2

# phase 3 (remove block comment)
<#

# get a list of required proxies
# first create initial session state needed for remoting
$iss =
[Management.Automation.Runspaces.InitialSessionState]::CreateRestricted("remo
teserver")

foreach ($proxy in $iss.Commands | where { $_.Visibility -eq "Public"})
{

    # bind proxy function
    Set-Item "function:global:$(($proxy.Name))" $proxy.Definition
}

# define public content

function Get-SystemInfo {
@{

Date = Get-Date
OSVersion = (Get-WmiObject Win32_OperatingSystem).Version
FreePhysicalMemory = (Get-WmiObject Win32_OperatingSystem).FreePhysicalMemory
LastBootUpTime = (Get-WmiObject
Win32_OperatingSystem).ConvertToDateTime((Get-WmiObject
Win32_OperatingSystem).LastBootUpTime)
FreeSpace = (Get-PSDrive c).Free
Culture = (Get-Culture).Name
WinUpdateServiceStatus = (Get-Service wuauerv).status
EventLog = Get-EventLog -LogName system -EntryType warning,error -Newest 5 -
Source 'Microsoft-Windows-WindowsUpdateClient' | select timegenerated,message
}
}

function Restart-WindowsUpdateService {
Restart-Service wuauerv -Force
}

#>
# end of phase 3

# phase 4 (insert this snippet before you create proxy commands)
<#

    $cmd = Get-Command -type cmdlet -ea silentlycontinue $proxy.name
    if ($cmd)
    {
        # define private alias to cmdlet
        # an alias that uses the module-qualified name of a cmdlet
        $a = Set-Alias "$($proxy.name)" "$($cmd.ModuleName)\$($cmd.Name)" -
PassThru

```

```
        $a.Visibility = "Private"
    }

#>
# end of phase 4
```

## demo.ps1

```
# what do we know about default remote session?

$default = New-PSSession core1; $default

# local and remote session run in different processes hosted in diff. hosts
$host.name
$pid
Invoke-Command -Session $default -ScriptBlock {$host.name}
Invoke-Command -Session $default -ScriptBlock {$pid}
icm -Session $default -script {gps | ? {$_ .id -eq $pid} | select
Id,ProcessName}

# a thread apartment state in a remote session is MTA by default
# even if a remote session is created in a shell that has been started using
STA
icm $default {[threading.thread]::currentthread}.apartmentstate}

# cmdlets for working with PSSessionConfiguration
Get-Command -Noun PSSessionConfiguration

# how to get registered PSSession configurations on a remote computer?
icm core1 {Get-PSSessionConfiguration}

Connect-WSMan core1
Push-Location
cd wsman:
dir
dir core1\plugin
Pop-Location

Get-WSManInstance winrm/config/plugin -Enumerate -cn core1 | ft name

# it would be easier if Get-PSSessionConfiguration supported ComputerName
parameter
# btw, the configurations are written in XML format in the Registry
# HKLM\Software\Microsoft\Windows\CurrentVersion\WSMan\Plugin

# let's run Get-PSSessionConfiguration locally
# we are interested in Name, StartupScript, and Permission properties
Get-PSSessionConfiguration

# only the members of local Administrators group can connect to a remote
session by default
# the configurations inherit the security descriptor from the RootSDDL
# the easiest way to look at permissions
Set-PSSessionConfiguration microsoft.powershell -ShowSecurityDescriptorUI

# let's move to a remote computer
```

```

# first, we will register the Demo.DeepDive configuration
# Register-PSSessionConfiguration -Name Demo.DeepDive
# and then we look at a security descriptor
# Set-PSSessionConfiguration Demo.DeepDive -ShowSecurityDescriptorUI

# PHASE 1 (on a remote computer)
# change our configuration using a startupscript that hides commands only
# Set-PSSessionConfiguration demo.deepdive -StartupScript
$pwd/constrained.ps1 -Force

$s = new-pssession -cn core1 -ConfigurationName demo.deepdive
icm $s {get-date}
icm $s {hostname}
icm $s {c:\demo\get-bios.ps1}
icm $s {& {get-date}}

# this behaviour is similar to modules
# when we can call a private function using the call operator and a module
context

import-module $pwd\testmodule.psml
Get-PublicFunction
Get-PrivateFunction
$m = get-module testmodule

$m | gm
& $m {Get-PrivateFunction}

# why do they all work? because we haven't constrained applications, scripts,
and language
# $ExecutionContext variable and its SessionState property to the rescue!
# we are particularly interested in the following properties:
# Applications, Scripts, and LanguageMode

icm $s {$ExecutionContext.SessionState}

# PHASE 2 (on a remote computer)
# add constrains for variables, applications, scripts, and language to
constrained.ps1 script
# Set-PSSessionConfiguration demo.deepdive -StartupScript
$pwd/ps1 -force

$s = new-pssession -cn core1 -ConfigurationName demo.deepdive
enter-pssession $s
import-pssession $s
# there are obviously some required commands
# how to get a minimal set of commands needed for interactive and implicit
remoting to work?

[Management.Automation.CommandMetaData]::GetRestrictedCommands("RemoteServer"
).GetEnumerator()

[Management.Automation.CommandMetaData]::GetRestrictedCommands("RemoteServer"
).GetEnumerator() |
ForEach-Object {$_.value}

```

```

# $_.value is of CommandMetaData type, so we can use that value to create a
proxy commands
# [Management.Automation.ProxyCommand]::Create($_.Value)
# or we can use InitialSessionState class that will create proxy commands
definitions for us

$iss =
[Management.Automation.Runspaces.InitialSessionState]::CreateRestricted("remo
teserver")

$iss.Commands | where { $_.Visibility -eq "Public" } | Format-Table name

# PHASE 3 (on a remote computer)
# create a required set of proxy commands
# and add our own set of functions (our remote service)
# Set-PSSessionConfiguration demo.deepdive -StartupScript
$pwd/constrained.ps1 -force

$s = new-pssession -cn core1 -ConfigurationName demo.deepdive
Invoke-Command $s {Get-Command}
Invoke-Command $s {Get-SystemInfo}

# why are we getting an error?
# Get-SystemInfo function uses one of the cmdlets that we proxied
# we changed/removed the parameters and acceptable arguments for Select-
Object cmdlet

# PHASE 4 (on a remote computer)
# we need to define the aliases that use the module-qualified name of a
cmdlet
# to bypass the constrained proxy function and call the cmdlet directly
# giving unconstrained access internally
# Set-PSSessionConfiguration demo.deepdive -StartupScript
$pwd/constrained.ps1 -force

$s = new-pssession -cn core1 -ConfigurationName demo.deepdive
Invoke-Command $s {Get-SystemInfo}
Import-PSSession $s
Get-SystemInfo

```