System

- Remove Ghost Devices
- View and Delete Local Profile List
- Extend the Windows RE Partition
- Get-AllEventLogsTimeFrame
- Ping With Log
- Send A Message To All Logged On Users
- Ping-WithTimestampAndLog

Remove Ghost Devices

removeghosts.ps1

```
<#
.SYNOPSIS
   Removes ghost devices from your system

.DESCRIPTION
    This script will remove ghost devices from your system. These are devices that are present but have an "InstallState" as false. These devices are typically shown as 'faded'
    in Device Manager, when you select "Show hidden and devices" from the view menu. This script has been tested on Windows 2008 R2 SP2 with PowerShell 3.0, 5.1, Server 2012R2
    with Powershell 4.0 and Windows 10 Pro with Powershell 5.1. There is no warranty with this script. Please use cautiously as removing devices is a destructive process without
    an undo.

.PARAMETER filterByFriendlyName
This parameter will exclude devices that match the partial name provided. This paramater needs to be specified in an array format for all the friendly names you want to be excluded.
"Intel" will match "Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz". "Loop" will match "Microsoft Loopback Adapter".

.PARAMETER narrowByFriendlyName
This parameter will include devices that match the partial name provided. This paramater needs to be specified in an array format for all the friendly names you want to be included.
"Intel" will match "Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz". "Loop" will match "Microsoft Loopback Adapter".

.PARAMETER filterByClass
This parameter will exclude devices that match the class name provided. This paramater needs to be specified in an array format for all the class names you want to be excluded.
This is an exact string match so "Disk" will not match "DiskDrive".

.PARAMETER narrowByClass
This parameter will include devices that match the class name provided. This paramater needs
```

```
to be specified in an array format for all the class names you want to be included.
This is an exact string match so "Disk" will not match "DiskDrive".

.PARAMETER listDevicesOnly
listDevicesOnly will output a table of all devices found in this system.

.PARAMETER listGhostDevicesOnly
listGhostDevicesOnly will output a table of all 'ghost' devices found in this system.

.PARAMETER force
If specified, each matching device will be removed WITHOUT any confirmation!

.EXAMPLE
Lists all devices
. "removeGhosts.ps1" -listDevicesOnly

.EXAMPLE
Save the list of devices as an object
$Devices = . "removeGhosts.ps1" -listDevicesOnly

.EXAMPLE
Lists all 'ghost' devices
. "removeGhosts.ps1" -listGhostDevicesOnly

.EXAMPLE
Lists all 'ghost' devices with a class of "Net"
. "removeGhosts.ps1" -listGhostDevicesOnly -narrowByClass Net

.EXAMPLE
Lists all 'ghost' devices with a class of "Net" AND a friendly name matching "Realtek"
. "removeGhosts.ps1" -listGhostDevicesOnly -narrowbyfriendlyname Realtek -narrowbyclass Net

.EXAMPLE
Save the list of 'ghost' devices as an object
$ghostDevices = . "removeGhosts.ps1" -listGhostDevicesOnly

.EXAMPLE
Remove all ghost devices EXCEPT any devices that have "Intel" or "Citrix" in their friendly
name
. "removeGhosts.ps1" -filterByFriendlyName @("Intel","Citrix")
```

```
.EXAMPLE
Remove all ghost devices that have "Intel" in their friendly name
. "removeGhosts.ps1" -narrowByFriendlyName Intel

.EXAMPLE
Remove all ghost devices EXCEPT any devices that are apart of the classes "LegacyDriver" or
"Processor"
. "removeGhosts.ps1" -filterByClass @("LegacyDriver","Processor")

.EXAMPLE
Remove all ghost devices EXCEPT for devices with a friendly name of "Intel" or "Citrix" or
with a class of "LegacyDriver" or "Processor"
. "removeGhosts.ps1" -filterByClass @("LegacyDriver","Processor") -filterByFriendlyName
@("Intel","Citrix")

.EXAMPLE
Remove all ghost network devices i.e. the ones with a class of "Net"
. "removeGhosts.ps1" -narrowByClass Net

.EXAMPLE
Remove all ghost devices without confirmation
. "removeGhosts.ps1" -Force

.NOTES
Permission level has not been tested.  It is assumed you will need to have sufficient rights
to uninstall devices from device manager for this script to run properly.
#>

Param(
  [array]$FilterByClass,
  [array]$NarrowByClass,
  [array]$FilterByFriendlyName,
  [array]$NarrowByFriendlyName,
  [switch]$listDevicesOnly,
  [switch]$listGhostDevicesOnly,
  [switch]$Force
)

#parameter futzing
```

```
$removeDevices = $true
if ($FilterByClass -ne $null) {
    write-host "FilterByClass: $FilterByClass"
}

if ($NarrowByClass -ne $null) {
    write-host "NarrowByClass: $NarrowByClass"
}

if ($FilterByFriendlyName -ne $null) {
    write-host "FilterByFriendlyName: $FilterByFriendlyName"
}

if ($NarrowByFriendlyName -ne $null) {
    write-host "NarrowByFriendlyName: $NarrowByFriendlyName"
}

if ($listDevicesOnly -eq $true) {
    write-host "List devices without removal: $listDevicesOnly"
    $removeDevices = $false
}

if ($listGhostDevicesOnly -eq $true) {
    write-host "List ghost devices without removal: $listGhostDevicesOnly"
    $removeDevices = $false
}

if ($Force -eq $true) {
    write-host "Each removal will happen without any confirmation: $Force"
}

function Filter-Device {
    Param (
        [System.Object]$dev
    )
    $Class = $dev.Class
    $FriendlyName = $dev.FriendlyName
    $matchFilter = $false

    if (($matchFilter -eq $false) -and ($null -ne $FilterByClass)) {
```

```
        foreach ($ClassFilter in $FilterByClass) {
            if ($ClassFilter -eq $Class) {
                Write-verbose "Class filter match $ClassFilter, skipping"
                $matchFilter = $true
                break
            }
        }
    }
    if (($matchFilter -eq $false) -and ($null -ne $NarrowByClass)) {
        $shouldInclude = $false
        foreach ($ClassFilter in $NarrowByClass) {
            if ($ClassFilter -eq $Class) {
                $shouldInclude = $true
                break
            }
        }
        $matchFilter = !$shouldInclude
    }
    if (($matchFilter -eq $false) -and ($null -ne $FilterByFriendlyName)) {
        foreach ($FriendlyNameFilter in $FilterByFriendlyName) {
            if ($FriendlyName -like '*'+$FriendlyNameFilter+'*') {
                Write-verbose "FriendlyName filter match $FriendlyName, skipping"
                $matchFilter = $true
                break
            }
        }
    }
    if (($matchFilter -eq $false) -and ($null -ne $NarrowByFriendlyName)) {
        $shouldInclude = $false
        foreach ($FriendlyNameFilter in $NarrowByFriendlyName) {
            if ($FriendlyName -like '*'+$FriendlyNameFilter+'*') {
                $shouldInclude = $true
                break
            }
        }
        $matchFilter = !$shouldInclude
    }
    return $matchFilter
}
```

```
function Filter-Devices {
    Param (
        [array]$devices
    )
    $filteredDevices = @()
    foreach ($dev in $devices) {
        $matchFilter = Filter-Device -Dev $dev
        if ($matchFilter -eq $false) {
            $filteredDevices += @($dev)
        }
    }
    return $filteredDevices
}
function Get-Ghost-Devices {
    Param (
        [array]$devices
    )
    return ($devices | Where-Object {$_.InstallState -eq $false} | Sort-Object -Property
FriendlyName)
}

# NOTE: White spaces are important in $setupapi for some reason!
$setupapi = @"
using System;
using System.Diagnostics;
using System.Text;
using System.Runtime.InteropServices;
namespace Win32
{
    public static class SetupApi
    {
         // 1st form using a ClassGUID only, with Enumerator = IntPtr.Zero
        [DllImport("setupapi.dll", CharSet = CharSet.Auto)]
        public static extern IntPtr SetupDiGetClassDevs(
           ref Guid ClassGuid,
           IntPtr Enumerator,
           IntPtr hwndParent,
           int Flags
        );
```

```
// 2nd form uses an Enumerator only, with ClassGUID = IntPtr.Zero
[DllImport("setupapi.dll", CharSet = CharSet.Auto)]
public static extern IntPtr SetupDiGetClassDevs(
   IntPtr ClassGuid,
   string Enumerator,
   IntPtr hwndParent,
   int Flags
);

[DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern bool SetupDiEnumDeviceInfo(
    IntPtr DeviceInfoSet,
    uint MemberIndex,
    ref SP_DEVINFO_DATA DeviceInfoData
);

[DllImport("setupapi.dll", SetLastError = true)]
public static extern bool SetupDiDestroyDeviceInfoList(
    IntPtr DeviceInfoSet
);
[DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern bool SetupDiGetDeviceRegistryProperty(
    IntPtr deviceInfoSet,
    ref SP_DEVINFO_DATA deviceInfoData,
    uint property,
    out UInt32 propertyRegDataType,
    byte[] propertyBuffer,
    uint propertyBufferSize,
    out UInt32 requiredSize
);
[DllImport("setupapi.dll", SetLastError = true, CharSet = CharSet.Auto)]
public static extern bool SetupDiGetDeviceInstanceId(
    IntPtr DeviceInfoSet,
    ref SP_DEVINFO_DATA DeviceInfoData,
    StringBuilder DeviceInstanceId,
    int DeviceInstanceIdSize,
    out int RequiredSize
);
```

```
        [DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
        public static extern bool SetupDiRemoveDevice(IntPtr DeviceInfoSet,ref SP_DEVINFO_DATA
DeviceInfoData);
    }
    [StructLayout(LayoutKind.Sequential)]
    public struct SP_DEVINFO_DATA
    {
       public uint cbSize;
       public Guid classGuid;
       public uint devInst;
       public IntPtr reserved;
    }
    [Flags]
    public enum DiGetClassFlags : uint
    {
        DIGCF_DEFAULT       = 0x00000001,  // only valid with DIGCF_DEVICEINTERFACE
        DIGCF_PRESENT       = 0x00000002,
        DIGCF_ALLCLASSES    = 0x00000004,
        DIGCF_PROFILE       = 0x00000008,
        DIGCF_DEVICEINTERFACE   = 0x00000010,
    }
    public enum SetupDiGetDeviceRegistryPropertyEnum : uint
    {
         SPDRP_DEVICEDESC          = 0x00000000, // DeviceDesc (R/W)
         SPDRP_HARDWAREID          = 0x00000001, // HardwareID (R/W)
         SPDRP_COMPATIBLEIDS           = 0x00000002, // CompatibleIDs (R/W)
         SPDRP_UNUSED0             = 0x00000003, // unused
         SPDRP_SERVICE             = 0x00000004, // Service (R/W)
         SPDRP_UNUSED1             = 0x00000005, // unused
         SPDRP_UNUSED2             = 0x00000006, // unused
         SPDRP_CLASS               = 0x00000007, // Class (R--tied to ClassGUID)
         SPDRP_CLASSGUID           = 0x00000008, // ClassGUID (R/W)
         SPDRP_DRIVER              = 0x00000009, // Driver (R/W)
         SPDRP_CONFIGFLAGS         = 0x0000000A, // ConfigFlags (R/W)
         SPDRP_MFG             = 0x0000000B, // Mfg (R/W)
         SPDRP_FRIENDLYNAME        = 0x0000000C, // FriendlyName (R/W)
         SPDRP_LOCATION_INFORMATION    = 0x0000000D, // LocationInformation (R/W)
         SPDRP_PHYSICAL_DEVICE_OBJECT_NAME = 0x0000000E, // PhysicalDeviceObjectName (R)
         SPDRP_CAPABILITIES        = 0x0000000F, // Capabilities (R)
         SPDRP_UI_NUMBER           = 0x00000010, // UiNumber (R)
```

```
        SPDRP_UPPERFILTERS        = 0x00000011, // UpperFilters (R/W)
        SPDRP_LOWERFILTERS        = 0x00000012, // LowerFilters (R/W)
        SPDRP_BUSTYPEGUID         = 0x00000013, // BusTypeGUID (R)
        SPDRP_LEGACYBUSTYPE           = 0x00000014, // LegacyBusType (R)
        SPDRP_BUSNUMBER           = 0x00000015, // BusNumber (R)
        SPDRP_ENUMERATOR_NAME         = 0x00000016, // Enumerator Name (R)
        SPDRP_SECURITY            = 0x00000017, // Security (R/W, binary form)
        SPDRP_SECURITY_SDS        = 0x00000018, // Security (W, SDS form)
        SPDRP_DEVTYPE             = 0x00000019, // Device Type (R/W)
        SPDRP_EXCLUSIVE           = 0x0000001A, // Device is exclusive-access (R/W)
        SPDRP_CHARACTERISTICS         = 0x0000001B, // Device Characteristics (R/W)
        SPDRP_ADDRESS             = 0x0000001C, // Device Address (R)
        SPDRP_UI_NUMBER_DESC_FORMAT       = 0X0000001D, // UiNumberDescFormat (R/W)
        SPDRP_DEVICE_POWER_DATA       = 0x0000001E, // Device Power Data (R)
        SPDRP_REMOVAL_POLICY          = 0x0000001F, // Removal Policy (R)
        SPDRP_REMOVAL_POLICY_HW_DEFAULT   = 0x00000020, // Hardware Removal Policy (R)
        SPDRP_REMOVAL_POLICY_OVERRIDE     = 0x00000021, // Removal Policy Override (RW)
        SPDRP_INSTALL_STATE           = 0x00000022, // Device Install State (R)
        SPDRP_LOCATION_PATHS          = 0x00000023, // Device Location Paths (R)
        SPDRP_BASE_CONTAINERID        = 0x00000024  // Base ContainerID (R)
    }
}
"@
Add-Type -TypeDefinition $setupapi

    #Array for all removed devices report
    $removeArray = @()
    #Array for all devices report
    $array = @()

    $setupClass = [Guid]::Empty
    #Get all devices
    $devs = [Win32.SetupApi]::SetupDiGetClassDevs([ref]$setupClass, [IntPtr]::Zero,
[IntPtr]::Zero, [Win32.DiGetClassFlags]::DIGCF_ALLCLASSES)

    #Initialise Struct to hold device info Data
    $devInfo = new-object Win32.SP_DEVINFO_DATA
    $devInfo.cbSize = [System.Runtime.InteropServices.Marshal]::SizeOf($devInfo)

    #Device Counter
```

```
    $devCount = 0
    #Enumerate Devices
    while([Win32.SetupApi]::SetupDiEnumDeviceInfo($devs, $devCount, [ref]$devInfo)) {

        #Will contain an enum depending on the type of the registry Property, not used but
required for call
        $propType = 0
        #Buffer is initially null and buffer size 0 so that we can get the required Buffer
size first
        [byte[]]$propBuffer = $null
        $propBufferSize = 0
        #Get Buffer size
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_FRIENDLYNAME, [ref]$propType, $propBuffer,
0, [ref]$propBufferSize) | Out-null
        #Initialize Buffer with right size
        [byte[]]$propBuffer = New-Object byte[] $propBufferSize

        #Get HardwareID
        $propTypeHWID = 0
        [byte[]]$propBufferHWID = $null
        $propBufferSizeHWID = 0
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_HARDWAREID, [ref]$propTypeHWID,
$propBufferHWID, 0, [ref]$propBufferSizeHWID) | Out-null
        [byte[]]$propBufferHWID = New-Object byte[] $propBufferSizeHWID

        #Get DeviceDesc (this name will be used if no friendly name is found)
        $propTypeDD = 0
        [byte[]]$propBufferDD = $null
        $propBufferSizeDD = 0
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_DEVICEDESC, [ref]$propTypeDD,
$propBufferDD, 0, [ref]$propBufferSizeDD) | Out-null
        [byte[]]$propBufferDD = New-Object byte[] $propBufferSizeDD

        #Get Install State
        $propTypeIS = 0
        [byte[]]$propBufferIS = $null
        $propBufferSizeIS = 0
```

```
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_INSTALL_STATE, [ref]$propTypeIS,
$propBufferIS, 0, [ref]$propBufferSizeIS) | Out-null
        [byte[]]$propBufferIS = New-Object byte[] $propBufferSizeIS

        #Get Class
        $propTypeCLSS = 0
        [byte[]]$propBufferCLSS = $null
        $propBufferSizeCLSS = 0
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_CLASS, [ref]$propTypeCLSS,
$propBufferCLSS, 0, [ref]$propBufferSizeCLSS) | Out-null
        [byte[]]$propBufferCLSS = New-Object byte[] $propBufferSizeCLSS
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_CLASS, [ref]$propTypeCLSS,
$propBufferCLSS, $propBufferSizeCLSS, [ref]$propBufferSizeCLSS)  | out-null
        $Class = [System.Text.Encoding]::Unicode.GetString($propBufferCLSS)

        #Read FriendlyName property into Buffer
        if(![Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_FRIENDLYNAME,
[ref]$propType, $propBuffer, $propBufferSize, [ref]$propBufferSize)){
            [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_DEVICEDESC,
[ref]$propTypeDD, $propBufferDD, $propBufferSizeDD, [ref]$propBufferSizeDD)  | out-null
            $FriendlyName = [System.Text.Encoding]::Unicode.GetString($propBufferDD)
            #The friendly Name ends with a weird character
            if ($FriendlyName.Length -ge 1) {
                $FriendlyName = $FriendlyName.Substring(0,$FriendlyName.Length-1)
            }
        } else {
            #Get Unicode String from Buffer
            $FriendlyName = [System.Text.Encoding]::Unicode.GetString($propBuffer)
            #The friendly Name ends with a weird character
            if ($FriendlyName.Length -ge 1) {
                $FriendlyName = $FriendlyName.Substring(0,$FriendlyName.Length-1)
            }
        }

        #InstallState returns true or false as an output, not text
```

```
        $InstallState = [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_INSTALL_STATE,
[ref]$propTypeIS, $propBufferIS, $propBufferSizeIS, [ref]$propBufferSizeIS)

        # Read HWID property into Buffer
        if(![Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_HARDWAREID,
[ref]$propTypeHWID, $propBufferHWID, $propBufferSizeHWID, [ref]$propBufferSizeHWID)){
            #Ignore if Error
            $HWID = ""
        } else {
            #Get Unicode String from Buffer
            $HWID = [System.Text.Encoding]::Unicode.GetString($propBufferHWID)
            #trim out excess names and take first object
            $HWID = $HWID.split([char]0x0000)[0].ToUpper()
        }

        #all detected devices list
        $device = New-Object System.Object
        $device | Add-Member -type NoteProperty -name FriendlyName -value $FriendlyName
        $device | Add-Member -type NoteProperty -name HWID -value $HWID
        $device | Add-Member -type NoteProperty -name InstallState -value $InstallState
        $device | Add-Member -type NoteProperty -name Class -value $Class
        if ($array.count -le 0) {
            #for some reason the script will blow by the first few entries without displaying
the output
            #this brief pause seems to let the objects get created/displayed so that they are
in order.
            Start-Sleep 1
        }
        $array += @($device)

        <#
        We need to execute the filtering at this point because we are in the current device
context
        where we can execute an action (eg, removal).
        InstallState : False == ghosted device
        #>
        if ($removeDevices -eq $true) {
            #we want to remove devices so let's check the filters...
```

```
            $matchFilter = Filter-Device -Dev $device

            if ($InstallState -eq $False) {
                if ($matchFilter -eq $false) {
                    $message  = "Attempting to remove device $FriendlyName"
                    $confirmed = $false
                    if (!$Force -eq $true) {
                        $question = 'Are you sure you want to proceed?'
                        $choices  = '&Yes', '&No'
                        $decision = $Host.UI.PromptForChoice($message, $question, $choices, 1)
                        if ($decision -eq 0) {
                            $confirmed = $true
                        }
                    } else {
                        $confirmed = $true
                    }
                    if ($confirmed -eq $true) {
                        Write-Host $message -ForegroundColor Yellow
                        $removeObj = New-Object System.Object
                        $removeObj | Add-Member -type NoteProperty -name FriendlyName -value
$FriendlyName
                        $removeObj | Add-Member -type NoteProperty -name HWID -value $HWID
                        $removeObj | Add-Member -type NoteProperty -name InstallState -value
$InstallState
                        $removeObj | Add-Member -type NoteProperty -name Class -value $Class
                        $removeArray += @($removeObj)
                        if([Win32.SetupApi]::SetupDiRemoveDevice($devs, [ref]$devInfo)){
                            Write-Host "Removed device $FriendlyName"  -ForegroundColor Green
                        } else {
                            Write-Host "Failed to remove device $FriendlyName" -
ForegroundColor Red
                        }
                    } else {
                        Write-Host "OK, skipped" -ForegroundColor Yellow
                    }
                } else {
                    write-host "Filter matched. Skipping $FriendlyName" -ForegroundColor
Yellow
                }
            }
```

```
    }
    $devcount++
}

#output objects so you can take the output from the script
if ($listDevicesOnly) {
    $allDevices = $array | Sort-Object -Property FriendlyName
    $filteredDevices = Filter-Devices -Devices $allDevices
    $filteredDevices | Format-Table
    write-host "Total devices found                : $($allDevices.count)"
    write-host "Total filtered devices found       : $($filteredDevices.count)"
    $ghostDevices = Get-Ghost-Devices -Devices $array
    $filteredGhostDevices = Filter-Devices -Devices $ghostDevices
    write-host "Total ghost devices found          : $($ghostDevices.count)"
    write-host "Total filtered ghost devices found : $($filteredGhostDevices.count)"
    return $filteredDevices | out-null
}

if ($listGhostDevicesOnly) {
    $ghostDevices = Get-Ghost-Devices -Devices $array
    $filteredGhostDevices = Filter-Devices -Devices $ghostDevices
    $filteredGhostDevices | Format-Table
    write-host "Total ghost devices found           : $($ghostDevices.count)"
    write-host "Total filtered ghost devices found  : $($filteredGhostDevices.count)"
    return $filteredGhostDevices | out-null
}

if ($removeDevices -eq $true) {
    write-host "Removed devices:"
    $removeArray  | Sort-Object -Property FriendlyName | Format-Table
    write-host "Total removed devices     : $($removeArray.count)"
    return $removeArray | out-null
}
```

View and Delete Local Profile List

delete_profile_with_gui_v2.ps1

```
########################################
# AUTHOR   : Ryan Mutschler
# DATE     : 8-15-2014
# EDIT     : 8-15-2014
# COMMENT  : GUI interface to delete user profiles from remote desktop session host
server
#
# VERSION  : 1    (Initial release)
# VERSION  : 2 - added support to select multiple users at once
########################################









#Setup script variables
#add computers to computers variable to search for profiles on those computers
[array] $Computers = "dcwipvmhsj001"
$log = "C:\temp\log.txt"
$date = Get-Date

#Reset variables
$selecteduser = ""
$profilelist = @()



Function SetupForm {
#Setup the form
[void] [System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
```

```
[void] [System.Reflection.Assembly]::LoadWithPartialName("System.Drawing")

$objForm = New-Object System.Windows.Forms.Form
$objForm.Text = "Select user(s)"
$objForm.Size = New-Object System.Drawing.Size(300,320)
$objForm.StartPosition = "CenterScreen"
$btnDelete = New-Object System.Windows.Forms.Button
$btnDelete.Location = New-Object System.Drawing.Size(120,240)
$btnDelete.Size = New-Object System.Drawing.Size(75,23)
$btnDelete.Text = "Delete Profile"
$objForm.Controls.Add($btnDelete)

#When a user clicks the delete button get the details of the logged in users and calls the
DeleteProfile function
$btnDelete.Add_Click({
#calls the delete profile function
    DeleteProfile
    }
    )

$CancelButton = New-Object System.Windows.Forms.Button
$CancelButton.Location = New-Object System.Drawing.Size(200,240)
$CancelButton.Size = New-Object System.Drawing.Size(75,23)
$CancelButton.Text = "Cancel"
$CancelButton.Add_Click({$objForm.Close()})
$objForm.Controls.Add($CancelButton)

$objLabel = New-Object System.Windows.Forms.Label
$objLabel.Location = New-Object System.Drawing.Size(10,20)
$objLabel.Size = New-Object System.Drawing.Size(280,20)
$objLabel.Text = "Please select user to delete profile:"
$objForm.Controls.Add($objLabel)

$objListBox = New-Object System.Windows.Forms.ListBox
$objListBox.Location = New-Object System.Drawing.Size(10,40)
$objListBox.Size = New-Object System.Drawing.Size(260,300)
$objListBox.Height = 180
$objListBox.SelectionMode = "MultiExtended"

#Run through each computer in the computers variable to compile a list of unique user accounts
```

```
across all servers
ForEach ($computer in $Computers) {
#use WMI to find all users with a profile on the servers
    Try{
        [array]$users = Get-WmiObject -ComputerName $computer Win32_UserProfile -filter
"LocalPath Like 'C:\\Users\\%'" -ea stop
        }
    Catch {
         Write-Warning "$($error[0]) "
        Break
        }

#compile the profile list and remove the path prefix leaving just the usernames
$profilelist = $profilelist + $users.localpath -replace "C:\\users\\"

#filter the user names to show only unique values left to prevent duplicates from profile
existing on multiple computers
$uniqueusers = $profilelist | Select-Object -Unique | Sort-Object
}

#adds the unique users to the combo box
ForEach($user in $uniqueusers) {
    [void] $objListBox.Items.Add($user)
}

$objForm.Controls.Add($objListBox)
$objForm.Topmost = $True
$objForm.Add_Shown({$objForm.Activate()})
[void] $objForm.ShowDialog()

}

Function DeleteProfile {
ForEach ($x in $objListBox.SelectedItems) {
    #Add the path prefix back to the selected user
    $selecteduser = $x
    $selectedUser = "C:\Users\$selecteduser"

    #This section reads through all the computers and deletes the profile from all the
computers - it catches any errors.
```

```
    ForEach ($computer in $Computers) {
        Try {
            (Get-WmiObject -ComputerName $computer Win32_UserProfile | Where-Object
{$_.LocalPath -eq $selecteduser}).Delete()
            Write-Host -ForegroundColor Green "$selecteduser has been deleted from $computer"
            Add-Content $log "$date $selecteduser profile has been deleted from $computer"
        }



        Catch [System.Management.Automation.MethodInvocationException]{
            Write-Host -ForegroundColor Red "ERROR: Profile is currently locked on $computer -
please use log off user script first"
            Add-Content $log "$date $selecteduser Profile is currently locked on $computer -
please use log off user script first"
        }

        Catch [System.Management.Automation.RuntimeException] {
            Write-Host -ForegroundColor Yellow -BackgroundColor Blue "INFO: $selecteduser
Profile does not exist on $computer"
            Add-Content $log "$date INFO: $selecteduser Profile does not exist on $computer"
        }

        Catch {
            Write-Host -ForegroundColor Red "ERROR: an unknown error occoured. The error
response was $error[0]"
            Add-Content $log "$date ERROR: an unknown error occoured. The error response was
$error[0]"
        }
     }
     }

#Add a label to say process is complete
$objLabel1 = New-Object System.Windows.Forms.Label
$objLabel1.Location = New-Object System.Drawing.Size(10,100)
$objLabel1.Size = New-Object System.Drawing.Size(280,20)
$objLabel1.Text = "Deletion complete, check log for more details."
$objForm.Controls.Add($objLabel1)
```

```
#Add a view log button to view the log file
$LogButton = New-Object System.Windows.Forms.Button
$LogButton.Location = New-Object System.Drawing.Size(50,150)
$LogButton.Size = New-Object System.Drawing.Size(75,23)
$LogButton.Text = "View Log"
$LogButton.Add_Click({Invoke-Item $log})
$objForm.Controls.Add($LogButton)

}

#Check script was run as admin
If (-NOT ([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.Principal.WindowsBuiltI
nRole] "Administrator"))
{
[System.Windows.Forms.MessageBox]::Show("It doesn't appear you have run this PowerShell
session with administrative rights, the script may not function correctly. If no users are
displayed please ensure you run the script again using administrive rights.")
}



#Start the form
SetupForm
```

```
$pc = qwinsta /server:SERVERNAME | select-string "Disc" | select-string -notmatch "services"

if ($pc)
{
  $pc| % {

  logoff ($_.tostring() -split ' +')[2] /server:SERVERNAME

  }
}
```

Not sure what this is anymore

```
gwmi win32_userprofile |

select @{LABEL=”last used”;EXPRESSION={$_.ConvertToDateTime($_.lastusetime)}},

LocalPath, SID
| ft -a | out-file "C:\Temp\log.txt"
```

Extend the Windows RE Partition

Resize_script.ps1

The sample script below can be used to increase the size of recovery partition to successfully update Windows Recovery Environment (WinRE). It is recommended to have 250MB of free space in the recovery partition for WinRE updates to install successfully. On devices that may not have adequate free space in the recovery partition, the sample script below can be used to extend the recovery partition by 250 MB.

Reboot your machine before you run the script. This is critical as there may be pending partition operations staged on your machine that will need to be finalized before the script can safely increase the WinRE partition size. After your machine reboots open Powershell as admin and run **mkdir <path to new backup directory>** to create a backup directory that the script may use in case of failure to restore the original partition. Note the location of this backup directory as the script will ask for your backup path. If you are deploying this at scale, you can bypass the script prompting by using the parameters

```
-SkipConfirmation $true -BackupFolder
```

For example,

```
Resize_script.ps1 -SkipConfirmation $true -BackupFolder c:\winre_backup
```

```
Param (
[Parameter(Mandatory=$false,HelpMessage="Skip confirmation")][bool]$SkipConfirmation=$false,
[Parameter(Mandatory=$true,HelpMessage="Path to backup old WinRE partition content
to")][string]$BackupFolder
)
# ----------------------------------
# Helper functions
# ----------------------------------
```

```
# Log message
function LogMessage([string]$message)
{
    $message = "$message"
    Write-Host $message
}
# Extract numbers from string
function ExtractNumbers([string]$str)
{
    $cleanString = $str -replace "[^0-9]"
    return [long]$cleanString
}
# Display partition info using fsutil
# Return an array, the first value is total size and the second value is free space
function DisplayPartitionInfo([string[]]$partitionPath)
{
    $volume = Get-WmiObject -Class Win32_Volume | Where-Object { $partitionPath -contains
$_.DeviceID }
    LogMessage("  Partition capacity: " +  $volume.Capacity)
    LogMessage("  Partition free space: " + $volume.FreeSpace)
    return $volume.Capacity, $volume.FreeSpace
}
# Display WinRE status
function DisplayWinREStatus()
{
    # Get WinRE partition info
    $WinREInfo = Reagentc /info
    foreach ($line in $WinREInfo)
    {
        $params = $line.Split(':')
        if ($params.Count -lt 2)
        {
            continue
        }
        if (($params[1].Trim() -ieq "Enabled") -Or (($params[1].Trim() -ieq "Disabled")))
        {
            $Status = $params[1].Trim() -ieq "Enabled"
            LogMessage($line.Trim())
        }
        if ($params[1].Trim() -like "\\?\GLOBALROOT*")
```

```
    {
        $Location = $params[1].Trim()
        LogMessage($line.Trim())
    }
}

return $Status, $Location
}
# ----------------------------------
# Main execution
# ----------------------------------
# Clear the error
$Error.Clear()
# ----------------------------------
# Examining the system to collect required info
# for the execution
# Need to check WinRE status, collect OS and WinRE
# partition info
# ----------------------------------
LogMessage("Start time: $([DateTime]::Now)")
LogMessage("Examining the system...")
$NeedShrink = $true
$NeedCreateNew = $false
$NeedBackup = $false
# Get WinRE partition info
$InitialWinREStatus = DisplayWinREStatus
$WinREStatus = $InitialWinREStatus[0]
$WinRELocation = $InitialWinREStatus[1]
if (!$WinREStatus)
{
LogMessage("Error: WinRE Disabled")
exit 1
}
# Get System directory and ReAgent xml file
$system32Path = [System.Environment]::SystemDirectory
LogMessage("System directory: " + $system32Path)
$ReAgentXmlPath = Join-Path -Path $system32Path -ChildPath "\Recovery\ReAgent.xml"
LogMessage("ReAgent xml: " + $ReAgentXmlPath)
if (!(Test-Path  $ReAgentXmlPath))
{
```

```
□LogMessage("Error: ReAgent.xml cannot be found")
□exit 1
}
# Get OS partition
LogMessage("")
LogMessage("Collecting OS and WinRE partition info...")
$OSDrive = $system32Path.Substring(0,1)
$OSPartition = Get-Partition -DriveLetter $OSDrive
# Get WinRE partition
$WinRELocationItems = $WinRELocation.Split('\\')
foreach ($item in $WinRELocationItems)
{
    if ($item -like "harddisk*")
□{
□□$OSDiskIndex = ExtractNumbers($item)
□}
□if ($item -like "partition*")
□{
□□$WinREPartitionIndex = ExtractNumbers($item)
□}
}
LogMessage("OS Disk: " + $OSDiskIndex)
LogMessage("OS Partition: " + $OSPartition.PartitionNumber)
LogMessage("WinRE Partition: " + $WinREPartitionIndex)
$WinREPartition = Get-Partition -DiskNumber $OSDiskIndex -PartitionNumber $WinREPartitionIndex
$diskInfo = Get-Disk -number $OSDiskIndex
$diskType = $diskInfo.PartitionStyle
LogMessage("Disk PartitionStyle: " + $diskType)
# Display WinRE partition size info
LogMessage("WinRE partition size info")
$WinREPartitionSizeInfo = DisplayPartitionInfo($WinREPartition.AccessPaths)
LogMessage("WinRE Partition Offset: " + $WinREPartition.Offset)
LogMessage("WinRE Partition Type: " + $WinREPartition.Type)
LogMessage("OS partition size: " + $OSPartition.Size)
LogMessage("OS partition Offset: " + $OSPartition.Offset)
$OSPartitionEnds = $OSPartition.Offset + $OSPartition.Size
LogMessage("OS partition ends at: " + $OSPartitionEnds)
LogMessage("WinRE partition starts at: " + $WinREPartition.Offset)
$WinREIsOnSystemPartition = $false
if ($diskType -ieq "MBR")
```

```
{
□if ($WinREPartition.IsActive)
□{
□□LogMessage("WinRE is on System partition")
□□$WinREIsOnSystemPartition = $true
□}
}
if ($diskType -ieq "GPT")
{
□if ($WinREPartition.Type -ieq "System")
□{
□□LogMessage("WinRE is on System partition")
□□$WinREIsOnSystemPartition = $true
□}
}
# Checking the BackupFolder parameter
if ($PSBoundParameters.ContainsKey('BackupFolder'))
{
□LogMessage("")
□LogMessage("Backup Directory: [" + $BackupFolder + "]")
□
□$Needbackup = $true
□
□if ($WinREIsOnSystemPartition)
□{
□□$Needbackup = $false
□□LogMessage("WinRE is on System partition which will be preserved. No need to backup content")
□}
□else
□{
□□if (Test-path $BackupFolder)
□□{
□□□$items = Get-ChildItem -Path $BackupFolder
□□□if ($items)
□□□{
□□□□LogMessage("Error: Existing backup directory is not empty")
□□□□exit 1
□□□}
□□}
□□else
```

```
□□{
□□□LogMessage("Creating backup directory...")
□□□try
□□□{
□□□□$item = New-Item -Path $BackupFolder -ItemType Directory -ErrorAction Stop
□□□□if ($item)
□□□□{
□□□□□LogMessage("Backup directory created")
□□□□}
□□□□else
□□□□{
□□□□□LogMessage("Error: Failed to create backup directory [" + $BackupFolder + "]")
□□□□□exit 1
□□□□}
□□□} catch
□□□{
□□□□LogMessage("Error: An error occurred: $_")
□□□□exit 1
□□□}
□□}
□}
}
# -----------------------------------
# Verify whether we meet requirements of execution
# - WinRE cannot be on OS partition for the extension
# - WinRE partition must be the next partition after OS partition
# - If WinRE partition already have >=250MB free space, no need to do repartition
# - If there is enough unallocated space to grow the WinRE partition size, skip shrinking OS
#
# However, if the WinRE partition is before the OS partition, there is no chance to extend it
# As a result, it's better to create a new WinRE partition after the OS partition
# -----------------------------------
# Perform a few checks
LogMessage("")
LogMessage("Verifying if the WinRE partition needs to be extended or not...")
if (!(($diskType -ieq "MBR") -Or ($diskType -ieq "GPT")))
{
□LogMessage("Error: Got an unexpected disk partition style: " +$diskType)
□exit 1
}
```

```
# WinRE partition must be after OS partition for the repartition
if ($WinREPartitionIndex -eq $OSPartition.PartitionNumber)
{
    LogMessage("WinRE and OS are on the same partition, should not perform extension")
    exit 0
}
$supportedSize = Get-PartitionSupportedSize -DriveLetter $OSDrive
# if there is enough free space, skip extension
if ($WinREPartitionSizeInfo[1] -ge 250MB)
{
    LogMessage("More than 250 MB of free space was detected in the WinRE partition, there is no need to extend the partition")
    exit 0
}
if ($WinREPartition.Offset -lt $OSPartitionEnds)
{
    LogMessage("WinRE partition is not after OS partition, cannot perform extension")
    LogMessage("Need to create a new WinRE partition after OS partition")
    $NeedCreateNew = $true
    $NeedShrink = $true

    # Calculate the size of repartition
    # Will create a new WinRE partition with current WinRE partition size + 250 MB
    # The OS partition size will be shrunk by the new WinRE partition size
    $targetWinREPartitionSize = $WinREPartitionSizeInfo[0] + 250MB
    $shrinkSize = [Math]::Ceiling($targetWinREPartitionSize / 1MB) * 1MB
    $targetOSPartitionSize = $OSPartition.Size - $shrinkSize
    if ($targetOSPartitionSize -lt $supportedSize.SizeMin)
    {
        LogMessage("Error: The target OS partition size after shrinking is smaller than the supported minimum size, cannot perform the repartition")
        exit 1
    }
}
else
{
    if ($WinREIsOnSystemPartition)
    {
        LogMessage("WinRE parititon is after the OS partition and it's also System partition")
        LogMessage("Error: Got unexpected disk layout, cannot proceed")
```

```
    exit 1
  }
  if (!($WinREPartitionIndex -eq ($OSPartition.PartitionNumber + 1)))
  {
    LogMessage("Error: WinRE partition is not right after the OS partition, cannot extend WinRE partition")
    exit 1
  }
  # Calculate the size of repartition
  # Will shrink OS partitition by 250 MB
  $shrinkSize = 250MB
  $targetOSPartitionSize = $OSPartition.Size - $shrinkSize
  $targetWinREPartitionSize = $WinREPartitionSizeInfo[0] + 250MB
  $UnallocatedSpace = $WinREPartition.Offset - $OSPartitionEnds;
  # If there is unallocated space, consider using it
  if ($UnallocatedSpace -ge 250MB)
  {
    $UnallocatedSpace = $WinREPartition.Offset - $OSPartitionEnds;
    LogMessage("Found unallocated space between OS and WinRE partition: " + $UnallocatedSpace)
    LogMessage("There is already enough space to extend WinRE partition without shrinking the OS partition")
    $NeedShrink = $false
    $targetOSPartitionSize = 0
  }
  else
  {
    $shrinkSize = [Math]::Ceiling((250MB - $UnallocatedSpace)/ 1MB) * 1MB
    if ($shrinkSize > 250MB)
    {
      $shrinkSize = 250MB
    }
    $targetOSPartitionSize = $OSPartition.Size - $shrinkSize
    if ($targetOSPartitionSize -lt $supportedSize.SizeMin)
    {
      LogMessage("Error: The target OS partition size after shrinking is smaller than the supported minimum size, cannot perform the repartition")
      exit 1
    }
  }
}
```

```
# ----------------------------------
# Report execution plan and ask for user confirmation to continue
# ----------------------------------
# Report the changes planned to be executed, waiting for user confirmation
LogMessage("")
LogMessage("Summary of proposed changes")
if ($NeedCreateNew)
{
    LogMessage("Note: WinRE partition is before OS partition, need to create a new WinRE partition
after OS partition")
    LogMessage("Will shrink OS partition by " + $shrinkSize)
    LogMessage("  Current OS partition size: " + $OSPartition.Size)
    LogMessage("  Target OS partition size after shrinking: " + $targetOSPartitionSize)
    LogMessage("New WinRE partition will be created with size: ", $targetWinREPartitionSize)
    if ($WinREIsOnSystemPartition)
    {
        LogMessage("Existing WinRE partition is also system partition, it will be preserved")
    }
    else
    {
        LogMessage("Existing WinRE partition will be deleted")
        LogMessage("  WinRE partition: Disk [" + $OSDiskIndex + "] Partition [" + $WinREPartitionInde
+ "]")
        LogMessage("  Current WinRE partition size: " + $WinREPartitionSizeInfo[0])
    }
}
else
{
    if ($NeedShrink)
    {
        LogMessage("Will shrink OS partition by " + $shrinkSize)
        LogMessage("  Current OS partition size: " + $OSPartition.Size)
        LogMessage("  Target OS partition size after shrinking: " + $targetOSPartitionSize)
        if ($UnallocatedSpace -ge 0)
        {
            LogMessage("Unallocated space between OS and WinRE partition that will be used towards the n
WinRE partition: " + $UnallocatedSpace)
        }
    }
    else
```

```
□{
□□LogMessage("Will use 250MB from unallocated space between OS and WinRE partition")
□}
□LogMessage("Will extend WinRE partition size by 250MB")
□LogMessage("  WinRE partition: Disk [" + $OSDiskIndex + "] Partition [" + $WinREPartitionIndex
+ "]")
□LogMessage("  Current WinRE partition size: " + $WinREPartitionSizeInfo[0])
□LogMessage("  New WinRE partition size:     " + $targetWinREPartitionSize)
□LogMessage("WinRE will be temporarily disabled before extending the WinRE partition and
enabled automatically in the end")
□if ($UnallocatedSpace -ge 100MB)
□{
□□LogMessage("Warning: More than 100MB of unallocated space was detected between the OS and
WinRE partitions")
□□LogMessage("Would you like to proceed by using the unallocated space between the OS and the
WinRE partitions?")
□}
}
if ($Needbackup)
{
□LogMessage("")
□LogMessage("The contents of the old WinRE partition will be backed up to [" + $BackupFolder +
"]")
}
LogMessage("")
LogMessage("Please reboot the device before running this script to ensure any pending
partition actions are finalized")
LogMessage("")
if ($SkipConfirmation)
{
□LogMessage("User chose to skip confirmation")
□LogMessage("Proceeding with changes...")
}
else
{
□$userInput = Read-Host -Prompt "Would you like to proceed? Y for Yes and N for No"
□□
□if ($userInput -ieq "Y")
□{
□□LogMessage("Proceeding with changes...")
```

```
□}
□elseif ($userInput -ieq "N")
□{
□□LogMessage("Canceling based on user request, no changes were made to the system")
□□exit 0
□}
□else
□{
□□LogMessage("Error: Unexpected user input: [" + $userInput + "]")
□□exit 0
□}
}
LogMessage("")
LogMessage("Note: To prevent unexpected results, please do not interrupt the execution or
restart your system")
# -----------------------------------
# Do the actual execution
# The main flow is:
# 1. Check whether ReAgent.xml has stage location and clear it for repartiion
# 2. Disable WinRE as WinRE partition will be deleted
# 3. Perform the repartition to create a larger WinRE partition
# 4. Re-enable WinRE
# -----------------------------------
LogMessage("")
# Load ReAgent.xml to clear Stage location
LogMessage("Loading [" + $ReAgentXmlPath + "] ...")
$xml = [xml](Get-Content -Path $ReAgentXmlPath)
$node = $xml.WindowsRE.ImageLocation
if (($node.path -eq "") -And ($node.guid -eq "{00000000-0000-0000-0000-000000000000}") -And
($node.offset -eq "0") -And ($node.id -eq "0"))
{
□LogMessage("Stage location info is empty")
}
else
{
□LogMessage("Clearing stage location info...")
□$node.path = ""
□$node.offset = "0"
□$node.guid= "{00000000-0000-0000-0000-000000000000}"
□$node.id="0"
```

```
# Save the change
LogMessage("Saving changes to [" + $ReAgentXmlPath + "]...")
$xml.Save($ReAgentXmlPath)
}
# Disable WinRE
LogMessage("Disabling WinRE...")
reagentc /disable
if (!($LASTEXITCODE -eq 0))
{
LogMessage("Warning: encountered an error when disabling WinRE: " + $LASTEXITCODE)
exit $LASTEXITCODE
}
# Verify WinRE is under C:\Windows\System32\Recovery\WinRE.wim
$disableWinREPath = Join-Path -Path $system32Path -ChildPath "\Recovery\WinRE.wim"
LogMessage("Verifying that WinRE wim exists in downlevel at default location")
if (!(Test-Path $disableWinREPath))
{
LogMessage("Error: Cannot find " + $disableWinREPath)

# Re-enable WinRE
LogMessage("Re-enabling WinRE on error...")
reagentc /enable
if (!($LASTEXITCODE -eq 0))
{
LogMessage("Warning: encountered an error when enabling WinRE: " + $LASTEXITCODE)
}
exit 1
}
# ----------------------------------
# Perform the repartition
# 1. Resize the OS partition
# 2. Delete the WinRE partition
# 3. Create a new WinRE partition
# ----------------------------------
LogMessage("Performing repartition to extend the WinRE partition ...")
# 1. Resize the OS partition
if ($NeedShrink)
{
LogMessage("Shrinking the OS partition to create a larger WinRE partition")
LogMessage("Resizing the OS partition to: [" + $targetOSPartitionSize + "]...")
```

```
Resize-Partition -DriveLetter $OSDrive -Size $targetOSPartitionSize
if ($Error.Count -gt 0) {
    LogMessage("Error: Resize-Partition encountered errors: " + $Error[0].Exception.Message)

    # Re-enable WinRE
    LogMessage("Re-enabling WinRE on error...")
    reagentc /enable
    if (!($LASTEXITCODE -eq 0))
    {
        LogMessage("Warning: encountered an error when enabling WinRE: " + $LASTEXITCODE)
    }
    exit 1
}
$OSPartitionAfterShrink = Get-Partition -DriveLetter $OSDrive
LogMessage("Target partition size: " + $targetOSPartitionSize)
LogMessage("Size of OS partition after shrinking: " + $OSPartitionAfterShrink.Size)
}
# 2. Delete the WinRE partition
LogMessage("")
if ($WinREIsOnSystemPartition)
{
LogMessage("Existing WinRE partition is System partition, skipping deletion")
}
else
{
# If requested by user, backup rest of the content on WinRE partition to backup directory
if ($Needbackup)
{
    $sourcePath = $WinREPartition.AccessPaths[0]
    LogMessage("Copying content on WinRE partition from [" + $sourcePath + "] to [" +
$BackupFolder + "]...")

    # Copy-Item may have access issue with certain system folders, enumerate the children items
and exlcude them
    $items = Get-ChildItem -LiteralPath $sourcePath -Force
    foreach ($item in $items)
    {
        if ($item.Name -ieq "System Volume Information")
        {
            continue
```

```
    }
    $sourceItemPath = Join-Path -Path $sourcePath -ChildPath $item.Name
    $destItemPath = Join-Path -Path $BackupFolder -ChildPath $item.Name
    try
    {
        LogMessage("Copying [" + $sourceItemPath + "] to [" + $destItemPath + "]...")
        Copy-Item -LiteralPath $sourceItemPath -Destination $destItemPath -Recurse -Force
    } catch
    {
        LogMessage("Error: An error occurred during copy: $_")
        exit 1
    }
  }

  LogMessage("Backup completed")
  LogMessage("")
 }
 LogMessage("Deleting WinRE partition: Disk [" + $OSDiskIndex + "] Partition [" +
$WinREPartitionIndex + "]...")
 Remove-Partition -DiskNumber $OSDiskIndex -PartitionNumber $WinREPartitionIndex -
Confirm:$false
 if ($Error.Count -gt 0) {
  LogMessage("Error: Remove-Partition encountered errors: " + $Error[0].Exception.Message)
  exit 1
 }
}
# A short sleep for the partition change
Sleep 5
# 3. Create a new WinRE partition
LogMessage("")
LogMessage("Creating new WinRE partition...")
LogMessage("Target size: " + $targetWinREPartitionSize)
if ($diskType -ieq "GPT")
{
 $partition = New-Partition -DiskNumber $OSDiskIndex -Size $targetWinREPartitionSize -GptType
"{de94bba4-06d1-4d40-a16a-bfd50179d6ac}"

 $newPartitionIndex = $partition.PartitionNumber
 # A short sleep to make sure the partition is ready for formatting
 Sleep 2
```

```
□LogMessage("Formating the partition...")
□$result = Format-Volume -Partition $partition -FileSystem NTFS -Confirm:$false
□if ($Error.Count -gt 0) {
□□LogMessage("Error: Format-Volume encountered errors: " + $Error[0].Exception.Message)
□□exit 1
□}
}
else
{
□#$partition = New-Partition -DiskNumber $OSDiskIndex -Size $targetWinREPartitionSize -MbrType
0x27
□$targetWinREPartitionSizeInMb = [int]($targetWinREPartitionSize/1MB)
□$diskpartScript =
@"
select disk $OSDiskIndex
create partition primary size=$targetWinREPartitionSizeInMb id=27
format quick fs=ntfs label="Recovery"
set id=27
"@
□$TempPath = $env:Temp
□$diskpartSciptFile = Join-Path -Path $TempPath -ChildPath
"\ExtendWinRE_MBR_PowershellScript.txt"
□
□LogMessage("Creating temporary diskpart script to create Recovery partition on MBR disk...")
□LogMessage("Temporary diskpart script file: " + $diskpartSciptFile)
□$diskpartScript | Out-File -FilePath $diskpartSciptFile -Encoding ascii
□
□LogMessage("Executing diskpart script...")
□try
□{
□□$diskpartOutput = diskpart /s $diskpartSciptFile
□□
□□if ($diskpartOutput -match "DiskPart successfully")
□□{
□□□LogMessage("Diskpart script executed successfully")
□□}
□□else
□□{
□□□LogMessage("Error executing diskpart script:" + $diskpartOutput)
□□□exit 1
```

```
    }
    LogMessage("Deleting temporary diskpart script file...")
    Remove-Item $diskpartSciptFile
  }
  catch
  {
    LogMessage("Error executing diskpart script: $_")
    exit 1
  }

  $vol = Get-Volume -FileSystemLabel "Recovery"
  $newPartitionIndex = (Get-Partition | Where-Object { $_.AccessPaths -contains $vol.Path }
).PartitionNumber
}
if ($Error.Count -gt 0)
{
  LogMessage("Error: New-Partition encountered errors: " + $Error[0].Exception.Message)
  exit 1
}
LogMessage("New Partition index: " + $newPartitionIndex)
# Re-enable WinRE
LogMessage("Re-enabling WinRE...")
reagentc /enable
if (!($LASTEXITCODE -eq 0))
{
  LogMessage("Warning: encountered an error when enabling WinRE: " + $LASTEXITCODE)
  exit $LASTEXITCODE
}
# In the end, Display WinRE status to verify WinRE is enabled correctly
LogMessage("")
LogMessage("WinRE Information:")
$FinalWinREStatus = DisplayWinREStatus
$WinREStatus = $FinalWinREStatus[0]
$WinRELocation = $FinalWinREStatus[1]
if (!$WinREStatus)
{
  LogMessage("Warning: WinRE Disabled")
}
$WinRELocationItems = $WinRELocation.Split('\\')
foreach ($item in $WinRELocationItems)
```

```
{
    if ($item -like "partition*")
    {
        $WinREPartitionIndex = ExtractNumbers($item)
    }
}
LogMessage("WinRE Partition Index: " + $WinREPartitionIndex)
$WinREPartition = Get-Partition -DiskNumber $OSDiskIndex -PartitionNumber $WinREPartitionIndex
$WinREPartitionSizeInfoAfter = DisplayPartitionInfo($WinREPartition.AccessPaths)
LogMessage("")
LogMessage("OS Information:")
$OSPartition = Get-Partition -DriveLetter $OSDrive
LogMessage("OS partition size: " + $OSPartition.Size)
LogMessage("OS partition Offset: " + $OSPartition.Offset)
if (!($WinREPartitionIndex -eq $newPartitionIndex))
{
    LogMessage("Warning: WinRE is installed to partition [" + $WinREPartitionIndex +"], but the
newly created Recovery partition is [" + $newPartitionIndex + "]")
}
LogMessage("End time: $([DateTime]::Now)")
if ($NeedBackup)
{
    LogMessage("")
    LogMessage("The contents of the old WinRE partition has been backed up to [" + $BackupFolder +
"]")
}
LogMessage("")
LogMessage("Successfully completed the operation")
```

Get-AllEventLogsTimeFrame

Make changes to the variables $startTime and $endTime for exact dates/times, otherwise utilize the commented out sections.

```
#example: get all logs in the last minute
if($computerName -eq "" -OR $null -eq $computerName)
{
  $computerName = $env:COMPUTERNAME
}
#gather the log names
$logNames = @()
$allLogNames = get-winevent -computerName $computerName -ListLog *
foreach($logName in $allLogNames)
{
  if($logName.recordcount -gt 0) #filter empty logs
  {
    $logNames += $logName
  }
}
#get the time range
$startTime = '3/3/2025 05:17:15'#(Get-date).AddMinutes(-1)
$endTime = '3/3/2025 05:18:00'#Get-date
#get the actual logs
$logs = Get-WinEvent -computerName $computerName -FilterHashtable @{
LogName=$logNames.logName; StartTime=$startTime; EndTime=$endTime}
#this makes Out-GridView show the full log properties
($logs | ConvertTo-Json | ConvertFrom-Json).syncroot | Export-Csv -Path .\events.csv -
NoTypeInformation
```

Ping With Log

```
@echo off

set /p host=host Address:
set logfile=Log_%host%.log

echo Target Host = %host% >%logfile%
for /f "tokens=*" %%A in ('ping %host% -n 1 ') do (echo %%A>>%logfile% && GOTO Ping)
:Ping
for /f "tokens=* skip=2" %%A in ('ping %host% -n 1 ') do (
    echo %date% %time:~0,2%:%time:~3,2%:%time:~6,2% %%A>>%logfile%
    echo %date% %time:~0,2%:%time:~3,2%:%time:~6,2% %%A
    timeout 1 >NUL
    GOTO Ping)
```

Send A Message To All Logged On Users

```
shutdown -r -t 600 -c "This system is restarting in 10 minutes to recover OS stability. Please logoff now to prevent data loss."
timeout /t 5
shutdown -a
```

Ping-WithTimestampAndLog

Ping With Timestamp and Log

> Remove line 1 from each code block to remove the logging to file.

The script below pings the target 10 times.

```
Start-Transcript -Force -Path "C:\temp\ping.log"
Test-Connection -Count 10 -ComputerName COMPUTERNAME | Format-Table
@{Name='TimeStamp';Expression={Get-Date}},Address,ProtocolAddress,ResponseTime
```

The script below pings the target the maximum number of times for Powershell Versions below 7.2.

```
Start-Transcript -Force -Path "C:\temp\ping.log"
Test-Connection -Count 2147483647 -ComputerName COMPUTERNAME | Format-Table
@{Name='TimeStamp';Expression={Get-Date}},Address,ProtocolAddress,ResponseTime
```

The script below pings the target indefinitely.

> Requires Powershell Version 7.2 at minimum.

```
Start-Transcript -Force -Path "C:\temp\ping.log"
Test-Connection -Repeat -ComputerName COMPUTERNAME | Format-Table
@{Name='TimeStamp';Expression={Get-Date}},Address,ProtocolAddress,ResponseTime
```