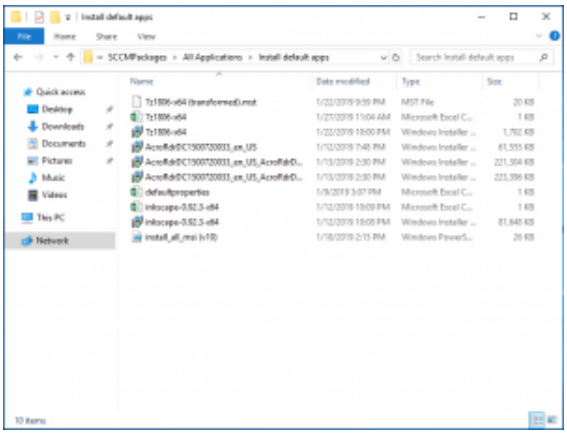
Install all the MSI installation files in a folder



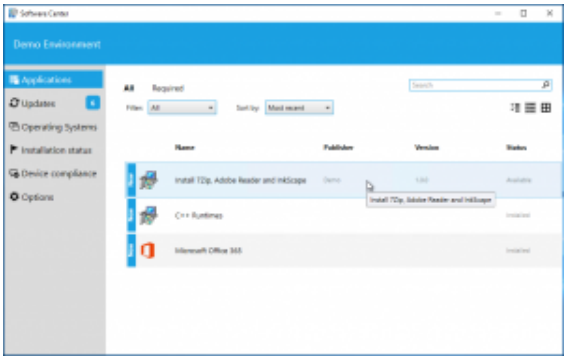
All the files in a folder

If you want to install a bunch of MSI files you put them in a folder and install them with a batch file. There is a downsize: you must modify the batch file for each situation. With this PowerShell script you can install all the MSI files in the folder, including applying transform and patch files. You can add your own properties in a csv settings file.

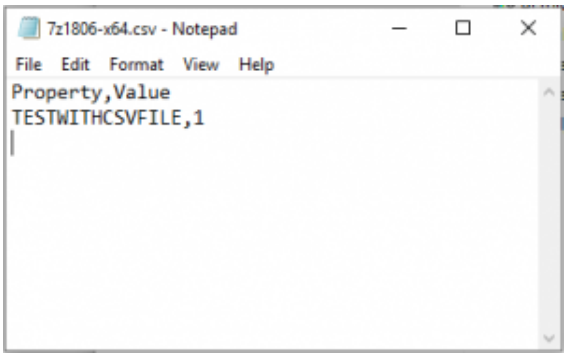
There are some parameters:

- **Install**: Use this swith to specify an installation.
- **Uninstall**: Use this swith to specify an uninstall.
- **MSIPath**: Specify the location where the MSI files are located.
- **Loglocation**: Specify the logfile location.

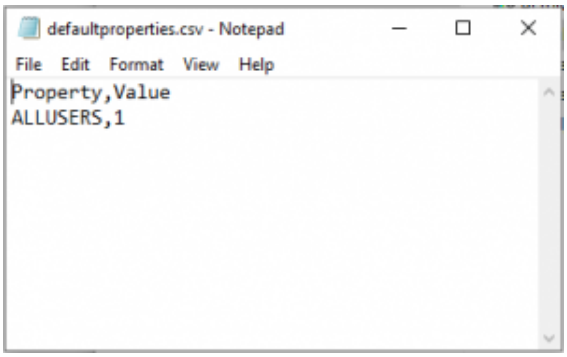
The default log location is C:\Windows\system32\LogFiles.



You can add a transform file.



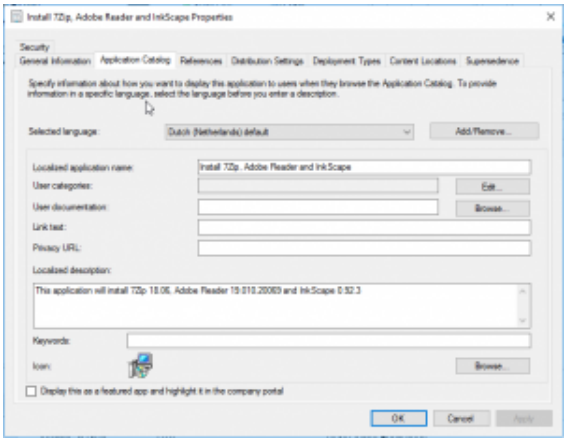
You can specify a settings file for each application.



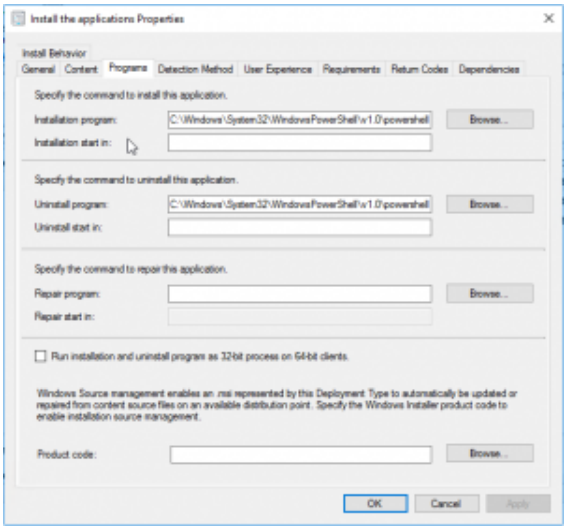
The default settings file is applied to all installations in the folder.

You can also use SCCM to install all the MSI files in the folder. Create an application with one deployment type. You can add all the product codes from each MSI file to identify a

successful installation.



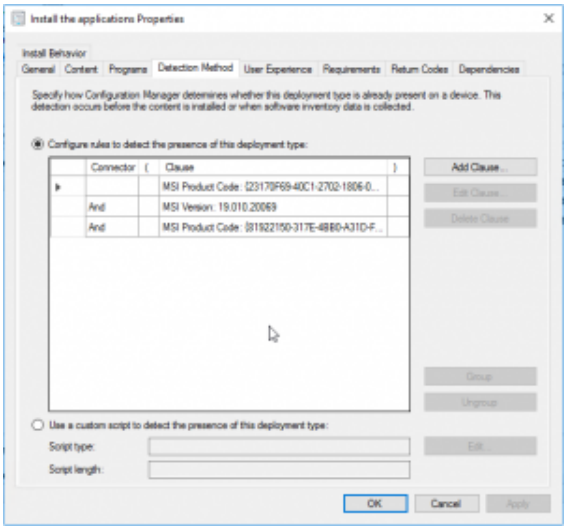
SCCM: The ‘Application Catalog’ tab.



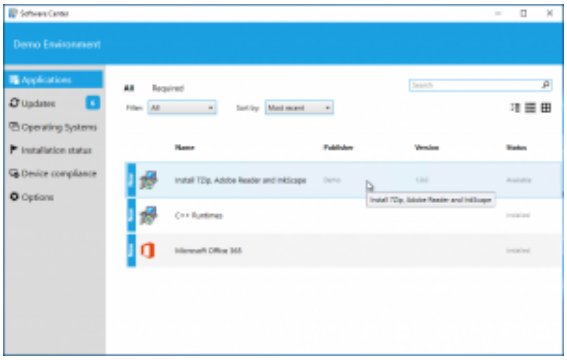
SCCM: Program

The **Install** line is:

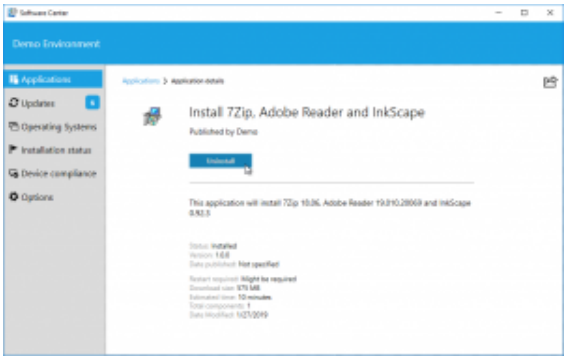
The **Uninstall** line is:



SCCM: Detection rule



Installation on the client.



Uninstall on the client.

A demonstration of this script can be found on my YouTube channel: [the script and AppV Repository]() or view it below:

The script:

```
<#

.SYNOPSIS
    Installs all MSI's in a folder. By default, it is the folder where the script is located, but you can specify another location.

.DESCRIPTION
    Installs all MSI's in a folder. By default, it is the folder where the script is located, but you can specify another location.
    If there is a MST that starts with the MSI name, then the MSI / MST combination will be installed.
    If there is a MSP that starts with the MSI name, then the MSI / MSP combination will be installed.

    Valid combinations:
    -> MSI file: Application_1.20.msi
    -> MST file: Application_1.20_transform.mst
    -> MSP file: Application_1.20_Update_to_1.40.msp
    -> CSV file: Application_1.20.csv

    Invalid combinations:
    -> MSI file: Application_1.20.msi
    -> MST file: Application_1.20.msi.mst     (.msi should be removed)
    -> MST file: Application_patch.msp        (not the full msi file name)

.EXAMPLE
    Installs all the MSIs that are in the folder where the script is located.
    ."\install_all_msi (v10).ps1"

.EXAMPLE
    Installs all the MSIs that are in the folder where the script is located.
    ."\install_all_msi (v10).ps1" -Install

.EXAMPLE
    Uninstalls all the MSIs that are in the folder
```

```
\\server\share\MSIs.
       ."\install_all_msi (v10).ps1" -Uninstall -MSIPath
\\server\share\MSIs

.EXAMPLE
      Uninstalls all the MSIs that are in the folder
\\server\share\MSIs. Logfiles are written to C:\Logs
       ."\install_all_msi (v10).ps1" -Uninstall -MSIPath
\\server\share\MSIs -LogLocation C:\Logs

.NOTES
    Author:  Willem-Jan Vroom
    Website: https://www.vroom.cc/
    Twitter: @TheStingPilot

 v0.1:
   * Initial version.

 v1.0:
   * Added: properties handler:
         - defaultproperties.csv -> will be applied to all MSI
packages or MSI / MST combination in the folder.
                                    This file can be in either
the location where the script is, or in the location
                                   where the MSI files are.
         - .csv     -> will be applied only to the given MSI
or MSI / MST combination.
                                    This file must be in the
same directory as the MSIs.
     This file must have the following header layout:
         Property,Value
         ALLUSERS,1
         ADDLOCAL,ALL
     The content may be different.
   * Install and uninstall switch added.
   * Check if user has admin rights. It throws up an error in
case not.
   * Added the command line options MSIPath and LogLocation.
   * Added patch support. The pathname must start with the
same name as the MSI.
   * Check if both install and uninstall switches are used.
```

```
   * Bugfix: error messages when there are quotes around the
MSI file name.

#>

[CmdLetBinding()]

param
 (
  # Use this switch to specify an installation.
  [Parameter(Mandatory=$False)]
  [Switch] $Install,

  # Use this switch to specify an uninstall.
  [Parameter(Mandatory=$False)]
  [Switch] $Uninstall,

  # Specify the location where the MSI files are located.
  [Parameter(Mandatory=$False)]
  [String] $MSIPath = "",

  # Specify the logfile location.
  [Parameter(Mandatory=$False)]
  [String] $LogLocation = ""
 )

#
================================================================
============================================================
# Function block
#
================================================================
============================================================

  Function CreateLogFile
  {

   <#
   .NOTES
================================================================
============================================================
```

```
   Created with:     Windows PowerShell ISE
   Created on:       9-January-2019
   Created by:       Willem-Jan Vroom
   Organization:
   Functionname:     CreateLogFile
=================================================================
=================================================================
   .SYNOPSIS

   This function creates the logfile

   #>

   param
    (
     [string] $LogFile
    )

   New-Item $LogFile -Force -ItemType File | Out-Null
  }

Function WriteToLog
 {
  <#
  .NOTES
=================================================================
=================================================================
  Created with:     Windows PowerShell ISE
  Created on:       9-January-2019
  Created by:       Willem-Jan Vroom
  Organization:
  Functionname:     WriteToLog
=================================================================
=================================================================
  .SYNOPSIS

  This function adds a line to the logfile

  #>

  param
```

```
     (
      [string] $LogFile,
      [string] $line
     )

   $timeStamp = (Get-Date).ToString('G').Replace("/","-")
   $line = $timeStamp + " - " + $line
   Add-Content -Path $LogFile -Value $line -Force
 }

 Function Import-PropertyFile
  {
   <#
   .NOTES
================================================================
================================================================
   Created with:     Windows PowerShell ISE
   Created on:       9-January-2019
   Created by:       Willem-Jan Vroom
   Organization:
   Functionname:     Import-PropertyFile
================================================================
================================================================
   .SYNOPSIS

   This function imports all the properties that are mentioned
in the given property-file

   #>

   param
    (
     [string] $PropertyFile
    )

   $arrItems      = @()
   $strProperties = ""

   if(Test-Path $PropertyFile)
    {
     $arrItems = @(Import-CSV $PropertyFile)
```

```
     if($arrItems.Count -ge 1)
      {
        WriteToLog -LogFile $strLogFile -line "The property
file $PropertyFile is applied."
       ForEach($objItem in $arrItems)
        {
         $strProperty    = $objItem.Property
         $strValue       = $objItem.Value
          $strLine        = $strProperty + "=" + $strValue + "
"
         $strProperties += $strLine
        }
       }
    }
    Return $strProperties
  }

Function Get-MSIFileInformation
 {

  <#
  .NOTES
================================================================
================================================================
  Created with:     Windows PowerShell ISE
  Created on:       9-January-2019
  Created by:       Willem-Jan Vroom
  Organization:
  Functionname:     Get-MSIFileInformation
================================================================
================================================================
  .SYNOPSIS

  This function reads the various properties from a MSI file.
        This    function    has    been    found    on
http://www.scconfigmgr.com/2014/08/22/how-to-get-msi-file-info
rmation-with-powershell/
   All credits, including the copyright go to Nickolaj
Andersen.

  #>
```

```
  param
   (
    [parameter(Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
    [System.IO.FileInfo]$Path,

    [parameter(Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
        [ValidateSet("ProductCode",  "ProductVersion",
"ProductName",   "Manufacturer",   "ProductLanguage",
"FullVersion")]
    [string]$Property
   )
  Process
   {
    try
     {
        # Read property from MSI database
            $WindowsInstaller = New-Object -ComObject
WindowsInstaller.Installer
                                          $MSIDatabase    =
$WindowsInstaller.GetType().InvokeMember("OpenDatabase",
"InvokeMethod", $null, $WindowsInstaller, @($Path.FullName,
0))
        $Query = "SELECT Value FROM Property WHERE Property =
'$($Property)'"
                                             $View     =
$MSIDatabase.GetType().InvokeMember("OpenView",
"InvokeMethod", $null, $MSIDatabase, ($Query))
             $View.GetType().InvokeMember("Execute",
"InvokeMethod", $null, $View, $null)
          $Record = $View.GetType().InvokeMember("Fetch",
"InvokeMethod", $null, $View, $null)
        $Value = $Record.GetType().InvokeMember("StringData",
"GetProperty", $null, $Record, 1)
        # Commit database and close view
            $MSIDatabase.GetType().InvokeMember("Commit",
"InvokeMethod", $null, $MSIDatabase, $null)
        $View.GetType().InvokeMember("Close", "InvokeMethod",
$null, $View, $null)
        $MSIDatabase = $null
```

```
        $View = $null

        # Return the value
        return $Value
     }
    catch
     {
      Write-Warning -Message $_.Exception.Message ; break
     }
  }
  End
  {
    # Run garbage collection and release ComObject
[System.Runtime.Interopservices.Marshal]::ReleaseComObject($Wi
ndowsInstaller) | Out-Null
    [System.GC]::Collect()
  }
}

Function Check-HasAdminRights
 {

  <#
  .NOTES
================================================================
================================================================
  Created with:     Windows PowerShell ISE
  Created on:       11-January-2019
  Created by:       Willem-Jan Vroom
  Organization:
  Functionname:     Check-HasAdminRights
================================================================
================================================================
  .SYNOPSIS

   This function checks if an user has admin rights. The
function returns $true or $false

  #>

        If    (([Security.Principal.WindowsPrincipal]
```

```
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([
Security.Principal.WindowsBuiltInRole] "Administrator"))
   {
    Return $True
   }
    else
   {
    Return $False
   }
 }

 Function Remove-TrailingCharacter
{

<#
.NOTES
===========================================================================
===========================================================================
=================
Created with:     Windows PowerShell ISE
Created on:       18-January-2019
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Remove-TrailingCharacter
===========================================================================
===========================================================================
=================
.SYNOPSIS

This function removes a trailing character from a string
#>

param
(
 [string] $RemoveCharacterFrom = "",
 [string] $Character           = ""
)

if($RemoveCharacterFrom.Length -gt 0)
 {
if(($RemoveCharacterFrom.SubString($RemoveCharacterFrom.Length
```

```
-1,1)) -eq $Character)
  {
                            $RemoveCharacterFrom         =
$RemoveCharacterFrom.Substring(0,$RemoveCharacterFrom.Length-1
)
  }
 }


 Return $RemoveCharacterFrom

}

 Function Add-TrailingCharacter
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Add-TrailingCharacter
================================================================
================================================================
================
.SYNOPSIS

This function adds a trailing backslash to a string
#>

param
(
 [string] $AddCharacterTo = "",
 [string] $Character      = ""
)

if($AddCharacterTo.Length -gt 0)
```

```
 {
    $AddCharacterTo  =  Remove-TrailingCharacter  -
RemoveCharacterFrom $AddCharacterTo -Character $([char]34)
  if(($AddCharacterTo.SubString($AddCharacterTo.Length-1,1)) -
ne $Character)
  {
   $AddCharacterTo = $AddCharacterTo + $Character
  }
 }
Else
 {
  $AddCharacterTo = $Character
 }

 Return $AddCharacterTo

}

Function Get-AllFilesWithPattern
 {

  <#
  .NOTES
================================================================
============================================================
  Created with:     Windows PowerShell ISE
  Created on:       13-January-2019
  Created by:       Willem-Jan Vroom
  Organization:
  Functionname:     Get-AllFilesWithPattern
================================================================
============================================================
  .SYNOPSIS

  Find all files in the given folder that matches a filter.

  #>

  param
   (
    [string] $FolderToLookIn,
```

```
    [string] $Pattern
   )

   $FolderToLookIn = Remove-TrailingCharacter -Character "\" -
RemoveCharacterFrom $FolderToLookIn

   $arrItems = @()
    $arrItems = Get-ChildItem -Path $FolderToLookIn -Filter
$Pattern
   $arrItems | Sort-Object -Property Name | Out-Null

   Return $arrItems
 }

Function Get-LastItemOfAnArryAndPutItInAString
 {

  <#
  .NOTES
================================================================
================================================================
  Created with:     Windows PowerShell ISE
  Created on:       13-January-2019
  Created by:       Willem-Jan Vroom
  Organization:
  Functionname:     Get-LastItemOfAnArryAndPutItInAString
================================================================
================================================================
  .SYNOPSIS

  Returns the last item of string.

  #>

  param
   (
    [string] $FileName,
    [string] $OldExtension,
    [string] $NewExtension,
    [string] $WhereToLook
   )
```

```
   $arrFiles    = @()
   $strFileName = ""
                $FilePattern     =     $FileName     -
Replace($OldExtension,$NewExtension)
    $arrFiles     = Get-AllFilesWithPattern -FolderToLookIn
$WhereToLook -Pattern $FilePattern
   if($arrFiles.Count -gt 0)
    {
     $strFileName = $arrFiles[-1].ToString()
    }

   Return $strFileName
}

#
=============================================================
============================================================
# End function block
#
=============================================================
============================================================

#
=============================================================
============================================================
# Define the variables.
#
=============================================================
============================================================

   $strCurrentDir               =  Split-Path  -parent
$MyInvocation.MyCommand.Definition

  if($MSIPath.Length -eq 0)
   {
    $MSIPath = $strCurrentDir
   }
  if($LogLocation.Length -eq 0)
   {
    $LogLocation = $Env:Windir + "\SYSTEM32\LogFiles"
   }
```

```
  $LogLocation              = Add-TrailingCharacter -
AddCharacterTo $LogLocation   -Character "\"
  $MSIPath                  = Add-TrailingCharacter -
AddCharacterTo $MSIPath       -Character "\"
  $strCurrentDir            = Add-TrailingCharacter -
AddCharacterTo $strCurrentDir -Character "\"

  $strTransform          = ""
  $strDefaultPropFile    = $MSIPath + "defaultproperties.csv"

  $strDefaultProperties  = ""
  $strActivity           = ""
  $strPatch              = ""
  $strSingleOrMultipleMSI = "MultipleMSI"

  $arrDefaultProperties  = @()
  $arrMSIFiles           = @()
  $arrMSPFiles           = @()
  $arrMSIFiles              = Get-AllFilesWithPattern -
FolderToLookIn $MSIPath -Pattern "*.msi"
  $numMSIFiles           = $arrMSIFiles.Count
  $numCounter            = 1

#
============================================================
============================================================
# Stop the script for a non admin user
#
============================================================
============================================================

  if(-not(Check-HasAdminRights))
  {
   Write-Error "The current user has no admin rights. Please
rerun  the  script  with  elevated  rights."  -Category
PermissionDenied
   Exit 999
  }

#
============================================================
```

```
=============================================================
# Create the log file location if not exists
#
=============================================================
=============================================================

  if(-not (Test-Path $LogLocation))
   {
    New-Item -Path $LogLocation -ItemType Directory -Force -
Confirm:$False | Out-Null
   }

#
=============================================================
=============================================================
# Define the logfile for the install or uninstall of all the
MSIs.
#
=============================================================
=============================================================

  $strLastPartOfFileName = " (" + (Get-Date).ToString('G') +
").log"
                       $strLastPartOfFileName          =
$strLastPartOfFileName.Replace(":","-").Replace("/","-")

  if($numMSIFiles -eq 1)
   {
    $strSingleOrMultipleMSI = "SingleMSI"
   }

  if($Install -or (-not $Uninstall))
   {
      $strLogFile                    =  $LogLocation  +
$strSingleOrMultipleMSI + $strLastPartOfFileName
    $strActivity           = "Installing MSIs in the folder
$MSIPath"
   }
    else
   {
    $strLogFile             = $LogLocation + "Uninstall" +
```

```
$strSingleOrMultipleMSI + $strLastPartOfFileName
    $strActivity           = "Uninstalling MSIs in the folder
$MSIPath"
   }

  CreateLogFile -LogFile $strLogFile

#
================================================================
================================================================
# Give an error message if both parameters install and
uninstall are used.
#
================================================================
================================================================

if($Install -and $Uninstall)
 {
  $strErrorMessage = "Both install and uninstall parameters
are mentioned. That is not possible. Only one of them should
be used."
                              WriteToLog              -line
"================================================================
================================================================" -
LogFile $strLogFile
  WriteToLog -line "FATAL ERROR!" -LogFile $strLogFile
  WriteToLog -line $strErrorMessage -LogFile $strLogFile
                              WriteToLog              -line
"================================================================
================================================================" -
LogFile $strLogFile
  Write-Error $strErrorMessage -Category InvalidArgument
  Exit 991
 }

#
================================================================
================================================================
# Write default settings to the logfile
#
================================================================
```

```
==========

                              WriteToLog             -line
"============================================================
============================================================" -
LogFile $strLogFile
   WriteToLog -line "Log location: $LogLocation" -LogFile
$strLogFile
  WriteToLog -line "MSI Path:     $MSIPath"     -LogFile
$strLogFile
  WriteToLog -line $($strActivity + ":")        -LogFile
$strLogFile

  ForEach ($objMSIFile in $arrMSIFiles)
   {
     WriteToLog -line " * $($objMSIFile.Name)" -LogFile
$strLogFile
   }
                              WriteToLog             -line
"============================================================
============================================================" -
LogFile $strLogFile

#
============================================================
============================================================
# In case of an installation:
# Define the default properties.
#
============================================================
============================================================

  if ($Install -or (-not $Uninstall))
   {
    $strDefaultProperties = Import-PropertyFile -PropertyFile
$strDefaultPropFile
   }
#
============================================================
============================================================
# Start the real installation or uninstall.
```

```
# The installation is skipped if a MSI has already been
installed.
# The uninstall is only done if the product has already been
installed.
#
================================================================
================================================================

  ForEach ($objMSIFile in $arrMSIFiles)
   {
    Write-Progress -Activity $($strActivity + ".") -Status
"Processing $objMSIFile." -PercentComplete ($numCounter /
$numMSIFiles * 100)
    $strMSIFileName    = $MSIPath + $objMSIFile
     $strProductName     = Get-MSIFileInformation -Path
$strMSIFileName -Property ProductName
      $strProductVersion = Get-MSIFileInformation -Path
$strMSIFileName -Property ProductVersion
     $strProductCode     = Get-MSIFileInformation -Path
$strMSIFileName -Property ProductCode
                  $strRegPathX64                          =
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\U
ninstall\"+$strProductCode
    $strRegPathX64     = $strRegPathX64 -replace(" ","")
                  $strRegPathX86                          =
$strRegPathX64.Replace("WOW6432Node\","")
    if($Install -or (-not $Uninstall))
     {
                                                          #
================================================================
================================================================
      # The install
                                                          #
================================================================
================================================================

      $strTransform      = ""
      $strPatch          = ""
                   $strMSTFileName             =   Get-
LastItemOfAnArryAndPutItInAString -FileName $objMSIFile -
OldExtension ".msi" -NewExtension "*.mst" -WhereToLook
```

```
$MSIPath
                    $strMSPFileName             =   Get-
LastItemOfAnArryAndPutItInAString -FileName $objMSIFile -
OldExtension ".msi" -NewExtension "*.msp" -WhereToLook
$MSIPath
                     $strPropFile                        =
$strMSIFileName.Replace("msi","csv")

                                                                #
================================================================
================================================
      # Apply a patch (msp file) (if available)
                                                                #
================================================================
================================================

      if($strMSPFileName.Length -ge 1)
       {
         $strPatch = " /update " + $([char]34) + $MSIPath +
$strMSPFileName + $([char]34)
        WriteToLog -LogFile $strLogFile -line "The patch file
'$strMSPFileName' has been found and is applied."
       }

                                                                #
================================================================
================================================
      # Apply a transform file (mst) (if available)
                                                                #
================================================================
================================================

      if($strMSTFileName.Length -ge 1)
       {
        $strMSTFileName = $MSIPath + $strMSTFileName
          $strTransform   = "TRANSFORMS=" + $([char]34) +
$strMSTFileName + $([char]34)+" "
        WriteToLog -LogFile $strLogFile -line "The transform
file '$strMSTFileName' has been found and is applied."
       }
```

```
         WriteToLog -line "Installing application: $strProductName"     -LogFile $strLogFile
                WriteToLog  -line  "ProductVersion: $strProductVersion" -LogFile $strLogFile

       if(-not ((Test-Path $strRegPathX64) -or (Test-Path $strRegPathX86)))
       {
        $strProperties    = " "
        $strProperties    = Import-PropertyFile -PropertyFile $strPropFile
         $strMSILogFile     = "/l*v " + $([char]34) + $LogLocation + $strProductName + " " + $strProductVersion +".log" + $([char]34)
        $strArguments     = "/i "   + $([char]34) + $MSIPath + $objMSIFile + $([char]34) + $strPatch + " /qb! " + $strDefaultProperties + $strProperties + $strTransform + $strMSILogFile

       $strArguments     = $strArguments  -replace("   ","")
        WriteToLog -line "Command that is run:       msiexec $($strArguments)" -LogFile $strLogFile
        $StartProcess = (Start-Process -FilePath "msiexec.exe" -ArgumentList $strArguments -Wait -PassThru)
        WriteToLog -line "Result: $($StartProcess.ExitCode)" -LogFile $strLogFile
                                    WriteToLog    -line "=========================================================================================================" -LogFile $strLogFile
       }
        else
       {
         WriteToLog -line "This application has already been installed, thus skipping." -LogFile $strLogFile
                                    WriteToLog    -line "=========================================================================================================" -LogFile $strLogFile
       }
     }
```

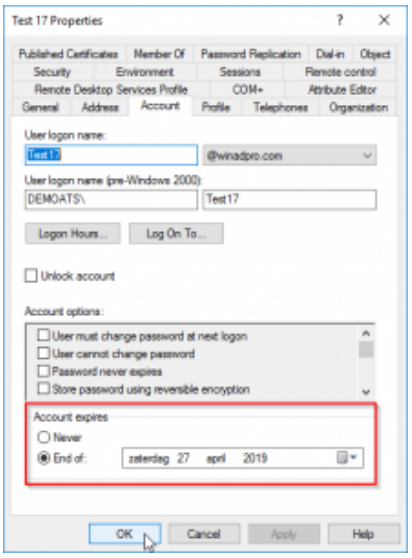
```
      else
     {
                                                                  #
================================================================
================================================
      # The uninstall
                                                                  #
================================================================
================================================

         WriteToLog -line "Uninstalling application:
$strProductName"     -LogFile $strLogFile
                WriteToLog  -line  "ProductVersion:
$strProductVersion" -LogFile $strLogFile
         if((Test-Path $strRegPathX64) -or (Test-Path
$strRegPathX86))
       {
          $strMSILogFile      = "/l*v " + $([char]34) +
$LogLocation + "Uninstall_"+ $strProductName + "  " +
$strProductVersion +".log" + $([char]34)
         $strArguments      = "/x "   + $strProductCode + "
/qb! " + $strMSILogFile

        $strArguments      = $strArguments  -replace("   ","")

        WriteToLog -line "Command that is run:         msiexec
$($strArguments)" -LogFile $strLogFile
        $StartProcess = (Start-Process -FilePath "msiexec.exe"
-ArgumentList $strArguments -Wait -PassThru)
        WriteToLog -line "Result: $($StartProcess.ExitCode)" -
LogFile $strLogFile

                                        WriteToLog    -line
"================================================================
================================================" -
LogFile $strLogFile
       }
        else
       {
          WriteToLog -line "This application has not been
installed, thus skipping." -LogFile $strLogFile
```

```
        WriteToLog -line " " -LogFile $strLogFile
       }
      }
     $numCounter++
    }

#
=============================================================
=============================================================
# Done!
#
=============================================================
=============================================================
```

Current version: [install_all_msi (v10).zip]

All expired user accounts and the accounts that are about to expire

A colleague of mine asked me to write a script with the Active Directory users with an account about to expire. So I created this script for him:



Account with an expiry date.

```
$arrUsers = @()
$arrOU = @("OU=Users,OU=OU1,DC=testdomain,DC=local,DC=lan","OU=Users,OU=OU2,DC=testdomain,DC=local,DC=lan")

ForEach($objOU in $arrOU)
 {
  $arrUsers += (Search-ADAccount -AccountExpiring -SearchBase $objOU -UsersOnly) | Select-Object "AccountExpirationDate","Name",@{Name="OU";Expression={($_."DistinguishedName" -split "=",3)[-1]}},"Enabled","LastLogonDate","LockedOut","ObjectClass","PasswordExpired","PasswordNeverExpires","SamAccountName","UserPrincipalName"
 }
$arrUsers | Sort-Object -Property "OU","Name" | Export-Csv "c:\temp\usersabouttoexpire.csv" -NoTypeInformation
```

It does what it should do, but not flexible. It can be used in one situation but needs modification for other situations. Also, you cannot specify the number of days when the account is about to expire.

So I created the script AccountsAboutToExpire (v10).ps1.

There are some parameters:

- **OUs**: Specify the OUs, seperated by a comma.
- **IncludeChildOUs**: Use this switch to include the child OU's.
- **NumberOfDaysToExpirationDate**: Specify the number of days in which the account expires. Default = 7

This script does not only give the accounts that are about to expire, but also the expired user accounts.

A demonstration of this script can be found on my YouTube channel: [the script and AppV Repository] or view it below:

The script:

```
<#
.SYNOPSIS
    Gives the accounts that about to expire within the given period.

.DESCRIPTION
    This script gives the accounts that are about to expire within the given period in a CSV file.

.EXAMPLE
    Reports all the accounts that are about to expire in the OUs  OU=Users,OU=OU1,DC=testdomain,DC=local,DC=lan  and OU=Users,OU=OU2,DC=testdomain,DC=local,DC=lan
          ."\AccountsAboutToExpire  V10).ps1"  -OUs "OU=Users,OU=OU1,DC=testdomain,DC=local,DC=lan","OU=Users,OU=OU2,DC=testdomain,DC=local,DC=lan"

.EXAMPLE
    Reports all the accounts that are about to expire in the OU DC=testdomain,DC=local,DC=lan, including the child OU's
          ."\AccountsAboutToExpire  V10).ps1"  -OUs "DC=testdomain,DC=local,DC=lan" -IncludeChildOUs

.EXAMPLE
```

```
    Reports all the accounts that are about to expire witin 60
days in the OU DC=testdomain,DC=local,DC=lan, including the
child OU's
          ."\AccountsAboutToExpire  V10).ps1"  -OUs
"DC=testdomain,DC=local,DC=lan"   -IncludeChildOUs   -
NumberOfDaysToExpirationDate 60

.NOTES
    Author:  Willem-Jan Vroom
    Website:
    Twitter: @TheStingPilot

v0.1:
   * Initial version.

v1.0:
   * Included:
      - All the expired user accounts
      - Sort on expiration date.
#>

[CmdLetBinding()]

param
(
# Specify the OUs, seperated by a comma.
[Parameter(Mandatory=$True)]
[string[]] $OUs,

# Use this switch to include the child OU's.
[Parameter(Mandatory=$False)]
[switch]  $IncludeChildOUs,

# Specify the number of days in which the account expires.
Default = 7
[Parameter(Mandatory=$False)]
[string]  $NumberOfDaysToExpirationDate=7
)

#
============================================================
```

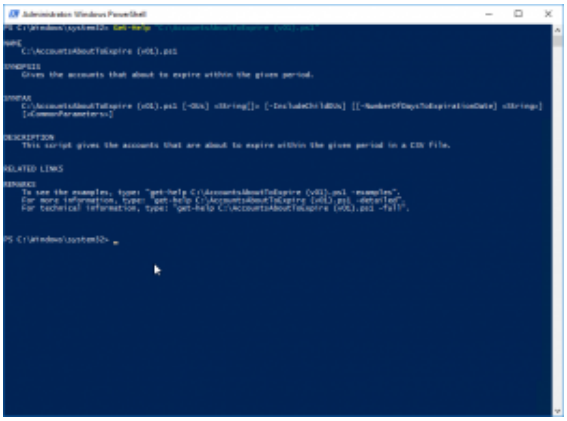
```
#================================================================
================
# Function block
#
#================================================================
================================================================
================

  Function Get-AccountsAboutToExpire
   {
    param
     (
      [string] $OU,
      [Switch] $InclChildOUs,
      [string] $NumDays
     )

    $arrItems       = @()
    $arrAboutToExp  = New-Object PSObject
    $arrExpired     = New-Object PSObject
    $strSearchScope = "OneLevel"

    if($InclChildOUs)
     {
      $strSearchScope = "SubTree"
     }

    Try
     {
       $arrAboutToExp  = (Search-ADAccount -AccountExpiring -
SearchBase $objOU -UsersOnly -SearchScope $strSearchScope -
TimeSpan       $NumDays)       |       Select-Object
"AccountExpirationDate","Name",@{Name="OU";Expression={($_."Di
stinguishedName"                                    -split
"=",3)[-1]}},@{Name="ExpiredAccount";Expression={$False}},"Ena
bled","LastLogonDate","LockedOut","PasswordExpired","PasswordN
everExpires"
      $arrExpired     = (Search-ADAccount -AccountExpired  -
SearchBase $objOU -UsersOnly -SearchScope $strSearchScope)
|                                           Select-Object
"AccountExpirationDate","Name",@{Name="OU";Expression={($_."Di
```

```
stinguishedName"                                    -split
"=",3)[-1]}},@{Name="ExpiredAccount";Expression={$True}},
"Enabled","LastLogonDate","LockedOut","PasswordExpired","Passw
ordNeverExpires"
      $arrItems      += $arrAboutToExp
      $arrItems      += $arrExpired
     }
      Catch
     {
      Write-Warning "The OU $OU has not been found."
     }

    Return $arrItems

   }

Function Check-HasAdminRights
 {

  <#
  .NOTES
================================================================
================================================================
  Created with:     Windows PowerShell ISE
  Created on:       11-January-2019
  Created by:       Willem-Jan Vroom
  Organization:
  Functionname:     Check-HasAdminRights
================================================================
================================================================
  .SYNOPSIS

   This function checks if an user has admin rights. The
function returns $true or $false

  #>

       If    (([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([
Security.Principal.WindowsBuiltInRole] "Administrator"))
   {
```

```
    Return $True
   }
    else
   {
    Return $False
   }
 }

#
================================================================
================================================================
================
# End function block
#
================================================================
================================================================
================

#
================================================================
============================================================
# Stop the script for a non admin user
#
================================================================
============================================================

  if(-not(Check-HasAdminRights))
  {
   Write-Error "The current user has no admin rights. Please
rerun the script with elevated rights." -Category
PermissionDenied
   Exit 999
  }

#
================================================================
================================================================
================
# Check if the module ActiveDirectory has been loaded.
#
================================================================
```

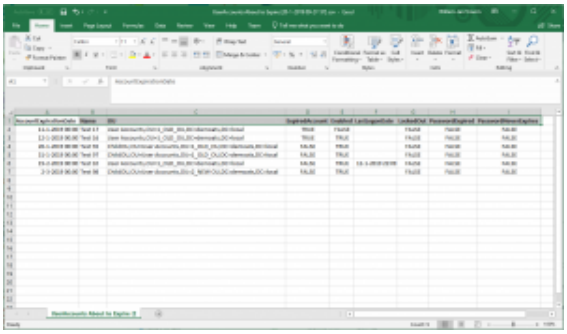
```
================================================================
================

  if(-not(Get-Module -ListAvailable ActiveDirectory))
   {
    Write-Warning "The module ActiveDirectory is not found.
Thus quitting."
    Exit 9
   }

#
================================================================
================================================================
================
# Define the variables
#
================================================================
================================================================
================

  $arrUsers      = @()
        $strCurrentDir    =    Split-Path    -parent
$MyInvocation.MyCommand.Definition
#
================================================================
================================================================
================
# Define the output file
#
================================================================
================================================================
================

  $strOutputFile = "UserAccounts About to Expire (" + (Get-
Date).ToString('G') + ").csv"
    $strOutputFile  =  $strOutputFile.replace(":","-
").Replace("/","-")
  $strOutputFile = $strCurrentDir + "\" + $strOutputFile

#
================================================================
```

```
# ================================================================
# Process the OUs.
# ================================================================

  ForEach($objOU in $OUs)
   {
    if($IncludeChildOUs)
     {
       $arrUsers += Get-AccountsAboutToExpire -OU $objOU -
NumDays $NumberOfDaysToExpirationDate -InclChildOUs
     }
      else
     {
       $arrUsers += Get-AccountsAboutToExpire -OU $objOU -
NumDays $NumberOfDaysToExpirationDate
     }
   }

# ================================================================
# Write the output to a csv file.
# ================================================================
          $arrUsers     |     Sort-Object     -Property
"AccountExpirationDate","OU","Name"    |    Export-Csv
$strOutputFile -NoTypeInformation
```



Get default help.

Get detailed help.



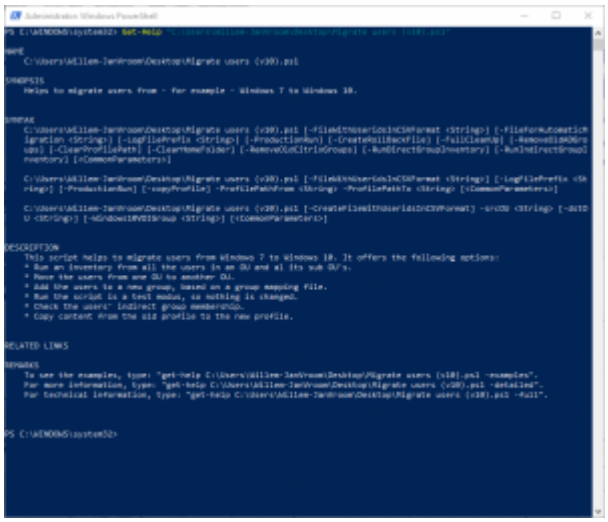
Output in Excel.

Current version: AccountsAboutToExpire (v10)

Previous versions:

- AccountsAboutToExpire (v01)

Improved script to migrate users from Windows 7 to Windows 10 automatically



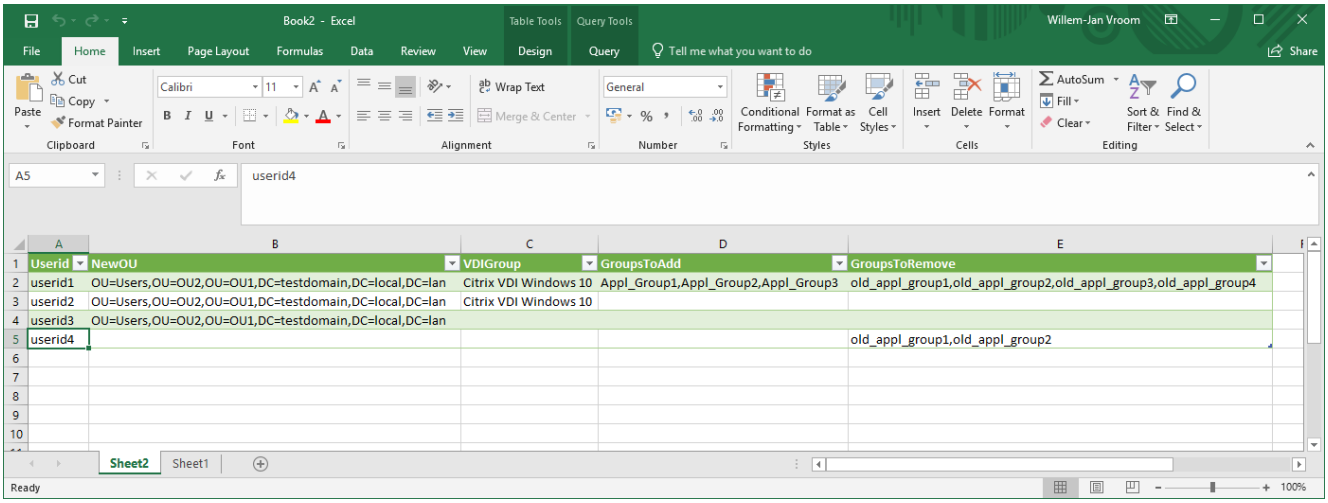
In my article, I wrote about a script to perform a user migration. In the meantime, that script has been improved to perform a more efficient and errorless migration.

The following improvements have been implemented:

- Provide a mappingsfile for an automatic migration. That mappingsfile contains both the old and the new group name.
- Perform a rollback in case of inpredicted results.
- In case of different profile locations: copy the old desktop and the old favorites to a new location.
- Improved help

To do this, I created a Powershell script.

The users who need to be migrated from Windows 7 to Windows 10 are in a csv file:



Userid	NewOU	VDIGroup	GroupsToAdd	GroupsToRemove
userid1	OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan	Citrix VDI Windows 10	Appl_Group1,Appl_Group2,Appl_Group3	old_appl_group1,old_appl_group2,old_appl_group3,old_appl_group4
userid2	OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan	Citrix VDI Windows 10		
userid3	OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan			
userid4				old_appl_group1,old_appl_group2

- The column **Userid** contains the userid
- The column **NewOU** contains the OU where the user should be moved to. This column can be empty.
- The colum **VDIGroup** contains the new VDI group. This column can be empty.
- The colum **GroupsToAdd** contains the groups the user should be added to. Split the groups with a comma. This column can be empty.
- The colum **GroupsToRemove** contains the groups the user should be removed from. Split the groups with a comma. This column can be empty.

An example:

```
"Userid","NewOU","VDIGroup","GroupsToAdd","GroupsToRemove"
"userid1","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan","Citrix VDI Windows 10","Appl_Group1,Appl_Group2,Appl_Group3","old_appl_group1,old_appl_group2,old_appl_group3,old_appl_group4"
"userid2","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan","Citrix VDI Windows 10","",""
"userid3","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan","","","",""
"userid4","","","","old_appl_group1,old_appl_group2"
```

Parameters for a migration:

- **FileWithUseridsInCSVFormat**: CSV Filename that contains

all the userids that should be migrated. Default = the script name, with the csv extension.

- **FileForAutomaticMigration**: CSV filename that contains the old and new group name for a fully automated migration. If not specified, then there is no automatic migration. Default is empty.
- **LogFilePrefix**: The name the logfile starts with. So the logfiles are grouped together.
- **ProductionRun**: Use this swith to update Active Directory. If not specified, the script is run in test mode.
- **CreateRollBackFile**: Use this switch to create a rollback file. That makes it possible to perform a rollback in case of unpredicted behavior.
- **FullCleanUp**: Use this swith to remove all unneeded groups.
- **RemoveOldADGroups**: Use this switch to remove the old AD groups in case of an automated migration.
- **ClearProfilePath**: Use this switch to clear the profile path.
- **ClearHomeFolder**: Use this swicth to clear the drive mapping and home folder.
- **RemoveOldCitrixGroups**: Use this switch to remove all the old Citrix groups from the users' account.
- **RunDirectGroupInventory**: Use this switch to display all the direct group membership in the result file.
- **RunIndirectGroupInventory**: Use this switch to display all the indirect group membership in the result file.

Parameters for an user inventory, based on an OU:

- **CreateFileWithUseridsInCSVFormat**: Use this switch to create a File with all the userids to migrate based on an OU.
- **srcOU**: The OU (distinguised name) where all the userids are found that needs to be migrated. If the OU contains spaces, then add quotes around the OU name.

- **dstOU**: The destination OU (distinguised name). If this one can is empty then the users are not moved to another OU. If the OU contains spaces, then add quotes around the OU name.
- **Windows10VDIGroup**: The new Windows 10 VDI group. If the VDI group contains spaces, then add quotes around it.

Parameters to copy the desktop and internet explorer favorites to another location:

- **FileWithUseridsInCSVFormat**: CSV Filename that contains all the userids that should be migrated. Default = the script name, with the csv extension.
- **ProfilePathFrom**: The old profile path (exluding userid). If the profile path contains spaces, then add quotes around it.
- **ProfilePathTo**: The new profile path (exluding userid). If the profile path contains spaces, then add quotes around it.
- **ProductionRun**: Use this switch to update Active Directory. If not specified, the script is run in test mode.

The script:

```
<#
.SYNOPSIS
    Helps to migrate users from - for example - Windows 7 to Windows 10.

.DESCRIPTION
     This script helps to migrate users from Windows 7 to Windows 10. It offers the following options:
    * Run an inventory from all the users in an OU and al its sub OU's.
    * Move the users from one OU to another OU.
    * Add the users to a new group, based on a group mapping file.
    * Run the script is a test modus, so nothing is changed.
```

```
    * Check the users' indirect group membership.
    * Copy content from the old profile to the new profile.

.EXAMPLE
     Perform the actions as described in the CSV Input File 'Userids-to-migrate.csv' in test mode:
     ."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv

.EXAMPLE
     Perform the actions as described in the CSV Input File 'Userids-to-migrate.csv' and add the groups as per mappingsfile 'MappingOldGroupToNewGroup.csv' in test mode:
     ."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv      -FileForAutomaticMigration MappingOldGroupToNewGroup.csv

.EXAMPLE
     Perform the actions as described in the CSV Input File 'Userids-to-migrate.csv' and add the groups as per mappingsfile 'MappingOldGroupToNewGroup.csv' in production:
     ."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv      -FileForAutomaticMigration MappingOldGroupToNewGroup.csv -ProductionRun

.EXAMPLE
     Perform the actions as described in the CSV Input File 'Userids-to-migrate.csv', add the groups as per mappingsfile 'MappingOldGroupToNewGroup.csv' in production and create a rollback file:
     ."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv      -FileForAutomaticMigration MappingOldGroupToNewGroup.csv    -ProductionRun    -CreateRollBackFile

.EXAMPLE
     Perform a rollback in production
     ."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat RBF_.csv -ProductionRun

.EXAMPLE
```

Perform the actions as described in the CSV Input File 'Userids-to-migrate.csv' and add the groups as per mappingsfile 'MappingOldGroupToNewGroup.csv' in production.

Also run both the direct- and indirect group membership inventory:

."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv -FileForAutomaticMigration MappingOldGroupToNewGroup.csv -ProductionRun -RunDirectGroupInventory -RunIndirectGroupInventory

.EXAMPLE

Perform the actions as described in the CSV Input File 'Userids-to-migrate.csv', add the groups as per mappingsfile 'MappingOldGroupToNewGroup.csv' and remove the old group in production:

."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv -FileForAutomaticMigration MappingOldGroupToNewGroup.csv -ProductionRun -RemoveOldADGroups

.EXAMPLE

Perform the actions as described in the CSV Input File 'Userids-to-migrate.csv', add the groups as per mappingsfile 'MappingOldGroupToNewGroup.csv', remove the old group in production and run the inventory for indirect group membership:

."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv -FileForAutomaticMigration MappingOldGroupToNewGroup.csv -ProductionRun -RemoveOldADGroups -RunIndirectGroupInventory

.EXAMPLE

Create a csv file with all the userids to migrate from the OU testdomain.local.lan\OU1\Old OU

."\Migrate users (v10).ps1" -CreateFileWithUseridsInCSVFormat -srcOU "OU=Old OU,OU=OU1,DC=testdomain,DC=local,DC=lan"

.EXAMPLE

Create a csv file with all the userids to migrate from the OU testdomain.local.lan\OU1\Ould OU. The new Citrix VDI group is 'Citrix10VDI'

```
                ."\Migrate users (v10).ps1" -CreateFileWithUseridsInCSVFormat -srcOU "OU=Old OU,OU=OU1,DC=testdomain,DC=local,DC=lan" -Windows10VDIGroup Citrix10VDI

.EXAMPLE
     Create a csv file with all the userids to migrate from the OU testdomain.local.lan\OU1\Ould OU. The new OU is testdomain.local.lan\OU1\OU2\Users. The new Citrix VDI group is 'Citrix10VDI'
                ."\Migrate users (v10).ps1" -CreateFileWithUseridsInCSVFormat -srcOU "OU=Old OU,OU=OU1,DC=testdomain,DC=local,DC=lan"  -dstOU "OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan" -Windows10VDIGroup Citrix10VDI

.EXAMPLE
     Migrate the IE favorites and the desktop folder from the old profile to the new profile in test mode:
      ."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv -copyProfile -ProfilePathFrom "\\server\share\w 7" -ProfilePathTo \\server\share\w10

.EXAMPLE
     Migrate the IE favorites and the desktop folder from the old profile to the new profile in production:
      ."\Migrate users (v10).ps1" -FileWithUseridsInCSVFormat Userids-to-migrate.csv -copyProfile -ProfilePathFrom "\\server\share\w 7" -ProfilePathTo \\server\share\w10 -ProductionRun

.NOTES
    Author:  Willem-Jan Vroom
    Website: https://www.vroom.cc/
    Twitter: @TheStingPilot

 v0.1:
   * Initial version.

 v0.2:
   * The logfilename has been changed.
```

```
 v0.3:
   * The logfile mentions 'RUNNING IN TEST MODE' in case the
swith -ProductionRun is not used.

 v1.0:
   * Introduction automatic migration.
   * Introduction Indirect Group Inventory
    * Automatic creation of the File with the userids to
migrate in CSV format.
   * Cleanup old Citrix groups
   * The results log file has a different lay-out
   * Parametersetnames
   * Improved help.
   * Copy desktop and favorites from the old profile to the
new profile.
   * Clear the homedir path in Active Directory.
   * Rollback scenario has been added.
#>

[CmdLetBinding()]

param
(
[CmdletBinding(DefaultParameterSetName = "Default")]

# CSV Filename that contains all the userids that should be
migrated. Default = the script name, with the csv extension.
# The filename with the users to be modified.
#
# This file has the following layout:
#
# "Userid","NewOU","VDIGroup","GroupsToAdd","GroupsToRemove"
#
"userid1","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=la
n","Citrix                  VDI                  Windows
10","Appl_Group1,Appl_Group2,Appl_Group3","old_appl_group1,old
_appl_group2,old_appl_group3,old_appl_group4"
#
"userid2","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=la
n","Citrix VDI Windows 10","",""
#
```

```
"userid3","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=la
n","","",""
# "userid4","","","","old_appl_group1,old_appl_group2"
#
# You can add more columns, but these columns are ignored.
[Parameter(Mandatory=$False,
           ParameterSetName="CopyProfile")]
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[String] $FileWithUseridsInCSVFormat = "",

# CSV filename that contains the old and new group name for a
fully automated migration. If not specified, then there is no
automatic migration. Defualt is empty.
#
# This file has the following layout:
#
# "OldGroup","NewGroup"
# "gg_appl_old1","appl_new1"
# "gg_appl_old2","appl_new2"
# "gg_appl_old1","appl_new3"
# "gg_appl_old4","appl_new4"
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[String] $FileForAutomaticMigration = "",

# The name the logfile starts with. So the logfiles are
grouped together.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Parameter(Mandatory=$False,
           ParameterSetName="CopyProfile")]

[String] $LogFilePrefix = "ZZZ-Logfile_",

# Use this swith to update Active Directory. If not specified,
the script is run in test mode.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Parameter(Mandatory=$False,
           ParameterSetName="CopyProfile")]
```

```
[Switch] $ProductionRun,

# Use this swith to create a rollback file. That makes it
possible to perform a rollback in case of unpredicted
behavior.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $CreateRollBackFile,

# Use this swith to remove all unneeded groups.
# If this switch is used then the following groups will be
removed from the users' account:
# * All gg_appl groups
# * WM-Users
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $FullCleanUp,

# Use this swith to remove the old AD groups in case of an
automated migration.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $RemoveOldADGroups,

# Use this swith to clear the profile path.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $ClearProfilePath,

# Use this swith to clear the drive mapping and home folder.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $ClearHomeFolder,

# Use this swith to remove all the old Citrix groups from the
users' account.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $RemoveOldCitrixGroups,

# Use this swith to display all the direct group membership in
```

```
the result file.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $RunDirectGroupInventory,

# Use this swith to display all the indirect group membership
in the result file.
[Parameter(Mandatory=$False,
           ParameterSetName="Default")]
[Switch] $RunIndirectGroupInventory,

# Use this swith to create a File with all the userids to
migrate based on an OU.
[Parameter(Mandatory=$False,
ParameterSetName="CreateFileWithUseridsInCSVFormat")]
[Switch] $CreateFileWithUseridsInCSVFormat,

# The OU (distinguised name) where all the userids are found
that needs to be migrated. If the OU contains spaces, then add
quotes around the OU name.
[Parameter(Mandatory=$True,
ParameterSetName="CreateFileWithUseridsInCSVFormat")]
[string] $srcOU,

# The destination OU (distinguised name). If this one can is
empty then the users are not moved to another OU. If the OU
contains spaces, then add quotes around the OU name.
[Parameter(Mandatory=$False,
ParameterSetName="CreateFileWithUseridsInCSVFormat")]
[string] $dstOU,

# The new Windows 10 VDI group. If the VDI group contains
spaces, then add quotes around it.
[Parameter(Mandatory=$False,
ParameterSetName="CreateFileWithUseridsInCSVFormat")]
[string] $Windows10VDIGroup="",

# Use this switch if you want to copy desktop and favorites
from the users' old profile to the users' new profile.
[Parameter(Mandatory=$False,
           ParameterSetName="CopyProfile")]
```

```
[switch] $copyProfile,

# The old profile path (exluding userid). If the profile path
contains spaces, then add quotes around it.
[Parameter(Mandatory=$True,
           ParameterSetName="CopyProfile")]
[string] $ProfilePathFrom="",

# The new profile path (exluding userid). If the profile path
contains spaces, then add quotes around it.
[Parameter(Mandatory=$True,
           ParameterSetName="CopyProfile")]
[string] $ProfilePathTo=""

)

#
================================================================
================================================================
================
# Function block
#
================================================================
================================================================
================

Function Write-EntryToResultsFile
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Write-EntryToResultsFile
================================================================
================================================================
```

```
================
.SYNOPSIS

This function adds the success or failure information to the
array that contains the log
information.

#>
param
 (
  [string] $strUserid,
  [string] $Result  = "",
  [string] $Action  = "",
  [string] $Message = ""
 )
  $Record                                    =  [ordered]
@{"Timestamp"="";"Username"  =  "";"Result"=  "";"Action"=
"";"Message"= ""}
 $Record."Timestamp"    = (Get-Date -UFormat "%a %e %b %Y
%X").ToString()
 $Record."Username"     = $strUserid
 $Record."Result"       = $Result
 $Record."Action"       = $Action
 $Record."Information"  = $Message
 $objRecord               = New-Object PSObject -Property
$Record
 $Global:arrTable      += $objRecord
}

Function Export-ResultsLogFileToCSV
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
```

```
Functionname:     Export-ResultsLogFileToCSV
=============================================================
=============================================================
================
.SYNOPSIS

This function writes the logfile content to a CSV file.

#>

 if($Global:arrTable.Count -gt 0)
   {
     $Global:arrTable | Export-Csv $strCSVLogFileSucces -
NoTypeInformation
   }
    Else
   {
    Write-Host "Something went wrong while writing the logfile
$strCSVLogFileSucces. Maybe nothing to report..."
   }
}

Function Export-RollBackFileToCSV
{

<#
.NOTES
=============================================================
=============================================================
================
Created with:     Windows PowerShell ISE
Created on:       29-November-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Export-RollBackFileToCSV
=============================================================
=============================================================
================
.SYNOPSIS

This function writes the rollbackfile content to a CSV file.
```

```
#>

 if($Global:arrTableWithRollBackRecords.Count -gt 0)
   {
      $Global:arrTableWithRollBackRecords | Export-Csv
$strCSVRollBackFile -NoTypeInformation
   }
    Else
   {
     Write-Host "Something went wrong while writing the file
$strCSVRollBackFile. Maybe nothing to report..."
   }
}

Function Remove-ProfilePathFromUserProfileInAD
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Remove-ProfilePathFromUserProfileInAD
================================================================
================================================================
================
.SYNOPSIS

This function clears the ProfilePath from AD.

#>

  param
   (
    [string] $strUserid
   )
```

```
    Write-EntryToResultsFile -strUserid $strUserid -Message
"Profilepath: $((Get-ADUser -Identity $strUserid -Properties
profilePath).profilePath)"  -Action  "Inventory"  -Result
"Success"

   Try
      {
                $strUserDN  =  (Get-ADUser  -Identity
$strUserid).distinguishedName
        Set-ADUser -Identity $strUserDN -Clear profilePath -
WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Clear profile path" -Result "Success"
      }
       Catch
      {
       Write-EntryToResultsFile -strUserid $strUserid -Message
$_.Exception.Message  -Action "Clear profile path" -Result
"Error"
       Continue
      }
}

Function Remove-HomeFolderPathFromUserProfileInAD
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Remove-HomeFolderPathFromUserProfileInAD
================================================================
================================================================
================
.SYNOPSIS
```

```
This function clears the Home Folder Path from AD.

#>

  param
   (
    [string] $strUserid
   )

    Write-EntryToResultsFile -strUserid $strUserid -Message
"Homedrive:     $((Get-ADUser -Identity $strUserid -Properties
homeDrive).homeDrive)" -Action "Inventory" -Result "Success"
    Write-EntryToResultsFile -strUserid $strUserid -Message
"Homedirectory: $((Get-ADUser -Identity $strUserid -Properties
homeDirectory).homeDirectory)" -Action "Inventory" -Result
"Success"

   Try
      {
                $strUserDN = (Get-ADUser -Identity
$strUserid).distinguishedName
       Set-ADUser -Identity $strUserDN -Clear homeDirectory -
WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Clear homedirectory path" -Result "Success"
      }
       Catch
      {
       Write-EntryToResultsFile -strUserid $strUserid -Message
$_.Exception.Message -Action "Clear homedirectory path" -
Result "Error"
       Continue
      }

    Try
      {
                $strUserDN = (Get-ADUser -Identity
$strUserid).distinguishedName
        Set-ADUser -Identity $strUserDN -Clear homeDrive -
WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
```

```
"Clear homedrive" -Result "Success"
      }
       Catch
      {
       Write-EntryToResultsFile -strUserid $strUserid -Message
$_.Exception.Message -Action "Clear homedrive" -Result "Error"
       Continue
      }
}

Function Move-ADUserToOtherOU
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Move-ADUserToOtherOU
================================================================
================================================================
================
.SYNOPSIS

This function moves the user to another OU.

#>

  param
   (
    [string] $strUserid,
    [string] $strDestinationOU
   )
   Try
     {
      if($strDestinationOU.Length -gt 0)
      {
```

```
                $strUserDN = (Get-ADUser -Identity
$strUserid).distinguishedName
         Move-ADObject -Identity $strUserDN -TargetPath
$strDestinationOU -WhatIf:(-not($ProductionRun)) -
Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Move AD User to OU $strDestinationOU." -Result "Success"
      }
     }
      Catch
     {
      Write-EntryToResultsFile -strUserid $strUserid -Message
$_.Exception.Message -Action "Move AD User to OU
$strDestinationOU." -Result "Error"
      Continue
     }
}

Function Add-ADMemberToGroup
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Add-ADMemberToGroup
================================================================
================================================================
================
.SYNOPSIS

This function adds a user to an AD group.

#>

  param
```

```
   (
    [string] $strUserid,
    [string] $strADGroupName
   )
   Try
     {
      if($strADGroupName.Length -gt 0)
      {
                $strUserDN  =  (Get-ADUser  -Identity
$strUserid).distinguishedName
        Add-ADGroupMember -Identity $strADGroupName -Members
$strUserDN -WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Add AD group $strADGroupName to user $strUserid." -Result
"Success"
       if($Global:strForRollBackGroupsToRemove.Length -eq 0)
       {
        $Global:strForRollBackGroupsToRemove = $strADGroupName
       }
       else
       {
              $Global:strForRollBackGroupsToRemove  =
$Global:strForRollBackGroupsToRemove + "," + $strADGroupName
       }
      }
     }
      Catch
     {
      Write-EntryToResultsFile -strUserid $strUserid -Message
$_.Exception.Message -Action "Add AD group $strADGroupName to
user $strUserid." -Result "Error"
      Continue
     }
}

Function Remove-ADMemberFromGroup
{

<#
.NOTES
============================================================
```

```
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Remove-ADMemberFromGroup
================================================================
================================================================
================
.SYNOPSIS

This function removes a user from an AD group.

#>

  param
   (
    [string] $strUserid,
    [string] $strADGroupName
   )
   Try
     {
      if($strADGroupName.Length -gt 0)
      {
                $strUserDN  =  (Get-ADUser  -Identity
$strUserid).distinguishedName
       Remove-ADGroupMember -Identity $strADGroupName -Members
$strUserDN -WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Remove AD group $strADGroupName from user $strUserid." -
Result "Success"
       if($Global:strForRollBackGroupsToAdd.Length -eq 0)
       {
        $Global:strForRollBackGroupsToAdd = $strADGroupName
       }
       else
       {
                 $Global:strForRollBackGroupsToAdd  =
$Global:strForRollBackGroupsToAdd + "," + $strADGroupName
       }
```

```
      }
     }
      Catch
     {
      Write-EntryToResultsFile -strUserid $strUserid -Message
$_.Exception.Message -Action "Remove AD group $strADGroupName
from user $strUserid." -Result "Error"
      Continue
     }
}

Function Remove-UserFromMultipleGroups
{

<#
.NOTES
===========================================================
===========================================================
=================
Created with:     Windows PowerShell ISE
Created on:       12-October-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Remove-UserFromMultipleGroups
===========================================================
===========================================================
=================
.SYNOPSIS

Delete multiple groups from the users' account. The group
names are in an array.

#>

  param
   (
    [string] $arrGroupsToSearchFor,
    [string] $strUserid
   )

      $arrGroups = @(Get-ADUser $strUserid -Properties
```

```
MemberOf).MemberOf
    forEach($objGroup in $arrGroups)
    {
    $strGroup=(Get-ADGroup $objGroup).Name
    forEach($objGroupMustContain in $arrGroupsToSearchFor)
     {
                                $strGroupMustContain     =
$objGroupMustContain.ToString().ToLower()
      if($strGroup.ToLower().IndexOf($strGroupMustContain) -eq
0)
       {
          Remove-ADMemberFromGroup -strUserid $strUserid -
strADGroupName $strGroup
       }
     }
    }


}

Function Migrate-ADUserToNewGroup
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       11-October-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Migrate-ADUserToNewGroup
================================================================
================================================================
================
.SYNOPSIS

Perform the translating from the old group name to the new
group name.
```

```
#>

  param
   (
    [string] $strUserid,
    [string] $strADGroupName
   )
   $bolOldGroupNameHasBeenFound = $False
       ForEach($objGroupNameForAutomaticMigration  in
$arrGroupNamesForAutomaticMigration)
    {
     Try
     {
                                          $strOldGroup      =
$objGroupNameForAutomaticMigration.OldGroup
                                          $strNewGroup      =
$objGroupNameForAutomaticMigration.NewGroup
     }
      Catch
     {
       Write-Host "There is something wrong with the CSV
filename  $FileForAutomaticMigration  while  processing
$strUserid."
      Exit 1
     }
     if ($strADGroupName -eq $strOldGroup)
     {
      $bolOldGroupNameHasBeenFound = $true
      if ($strNewGroup.Length -gt 0)
       {
           Add-ADMemberToGroup -strUserid $strUserid -
strADGroupName $strNewGroup
        if ($RemoveOldADGroups)
         {
           Remove-ADMemberFromGroup -strUserid $strUserid -
strADGroupName $strOldGroup
         }
       }
        Else
       {
        Write-EntryToResultsFile -strUserid $strUserid -Action
```

```
"Migrate users" -Message "No new groupname specified for $strADGroupName in $FileForAutomaticMigration." -Result "Error"
       }
     }

    }
    if(-not($bolOldGroupNameHasBeenFound))
    {
      Write-EntryToResultsFile -strUserid $strUserid -Action "Migrate users" -Message "The group $strADGroupName has not been found in $FileForAutomaticMigration." -Result "Error"
    }
  }

Function Find-InArry
{
<#
.NOTES
================================================================================================================
Created with:     Windows PowerShell ISE
Created on:       17-October-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Find-InArry
================================================================================================================
.SYNOPSIS

This function checks two arrays for content. I had to write my own function as the build-in functions like Compare-Object or $arrOnlyIndirectMembers | Where {$arrDirectAndIndirectMembers -notcontains $arrDirectMembers} did not did their work properly.

The swith 'NotContains' can be $True or $False. If not specified on the command line, then it is $False.
```

```
#>


param
(
 [string[]] $SearchIn,
 [string[]] $LookFor,
 [switch] $NotContains
)

$Contains = -not $NotContains

$tmpArray = @()
ForEach ($objSearchIn in $SearchIn)
 {
  $bolFound = $false
  ForEach ($objLookFor in $LookFor)
  {
   if($objSearchIn -eq $objLookFor)
    {
     $bolFound = $True
     if ($bolFound -and $Contains)
     {
      $tmpArray+=$objLookFor
     }
    }
  }
  if (-not($bolFound) -and -not $Contains)
  {
   $tmpArray+=$objSearchIn
  }
 }
 Return $tmpArray
}

Function Show-IndirectGroupsInResultsFile
{

<#
.NOTES
============================================================
```

```
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       18-October-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Show-IndirectGroupsInResultsFile
================================================================
================================================================
================
.SYNOPSIS

This function shows only the indirect group membership in the
results file.

#>

  param
   (
    [string] $strUserid
   )

    $dn                                    = (Get-ADUser
$strUserid).DistinguishedName
   $arrOnlyDirectMembers        = @()
   $arrDirectAndIndirectMembers = @()
   $arrOnlyIndirectMembers      = @()
    $arrDirectAndIndirectMembers = Get-ADGroup   -LDAPFilter
("(member:1.2.840.113556.1.4.1941:={0})" -f $dn)    | Select-
Object sAMAccountName | Sort-Object sAMAccountName
       $arrOnlyDirectMembers                    =   Get-
ADPrincipalGroupMembership                          $dn
| Select-Object sAMAccountName | Sort-Object sAMAccountName
    $arrOnlyIndirectMembers       = Find-InArry -SearchIn
$arrDirectAndIndirectMembers   -NotContains   -LookFor
$arrOnlyDirectMembers

   $arrOnlyIndirectMembers | ForEach($_) {
    $tmpValue = $_
    $tmpValue = $TmpValue -Replace("@{sAMAccountName=","")
    $tmpValue = $TmpValue -Replace("}","")
```

```
     Write-EntryToResultsFile -strUserid $strUserid -Result
"Information" -Action "Indirect group member" -Message
$TmpValue
   }

}

Function Show-DirectGroupsInResultsFile
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       15-November-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Show-DirectGroupsInResultsFile
================================================================
================================================================
================
.SYNOPSIS

This function shows only the direct group membership in the
results file.

#>

  param
   (
    [string] $strUserid
   )

   $arrOnlyDirectMembers        = @()
       $arrOnlyDirectMembers                    =  Get-
ADPrincipalGroupMembership              ((Get-ADUser
$strUserid).DistinguishedName)  | Select-Object sAMAccountName
| Sort-Object sAMAccountName
   $arrOnlyDirectMembers | ForEach($_) {
```

```
    $tmpValue = $_.sAMAccountName
     Write-EntryToResultsFile -strUserid $strUserid -Result
"Information" -Action "Direct group member" -Message $TmpValue
   }

}

Function Create-FileWithUseridsInCSVFormat
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       18-October-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Create-FileWithUseridsInCSVFormat
================================================================
================================================================
================
.SYNOPSIS

This function cretes the file with all the userids to migrate,
based on OU.

#>
  Param
    (
     [string] $srcOU ="",
     [string] $dstOU= "",
     [string] $Windows10VDIGroup= ""
    )

     $strBaseFileName = "UseridsToMigrate (" + (Get-
Date).ToString('G') + ").csv"
   $strBaseFileName = $strBaseFileName -replace ":","-"
   $strBaseFileName = $strBaseFileName -replace "/","-"
   $strFileName     = $strCurrentPath + "\" + $strBaseFileName
```

```
   if($strFileName.Length -gt 260)
   {
    $valLength       = ($strCurrentPath.Length)-4
      $strBaseFileName = ($strBaseFileName.Substring(0,260-
$valLength)) + ".csv"
       $strFileName        =  $strCurrentPath  +  "\"  +
$strBaseFileName
   }

   $arrTableWithUserids = @()
    $arrUserids           = (Get-ADUser -Filter {Enabled -eq
"True"} -SearchScope Subtree -SearchBase $srcOU)
   $valCounter          = 1

   $Users = ForEach ($objUserid in $arrUserids)
   {
     $strUserDetails = Get-ADUser $objUserid -Properties
Name,sAMAccountName,department,cn, DistinguishedName, mail
    Write-Progress -Activity "Create CSV File with the userids
for    migration"    -Status    "Processing    user
$($strUserDetails.cn)."  -PercentComplete  ($valCounter  /
$arrUserids.Count * 100)
    $UserRecord                    = [ordered] @{"Userid" =
"";"Full  name"=  "";"Mail"="";"Department"=  "";"CurrentOU"=
"";"NewOU"="";"VDIGroup"=          "";"GroupsToAdd"=
"";"GroupsToRemove"= ""}
         $UserRecord."Userid"                          =
$strUserDetails.sAMAccountName
    $UserRecord."Full name"       = $strUserDetails.cn
    $UserRecord."Mail"            = $strUserDetails.mail
    $UserRecord."Department"      = $strUserDetails.department
        $UserRecord."CurrentOU"              =  "OU="+
($strUserDetails.DistinguishedName -split "=",3)[-1]
    $UserRecord."NewOU"           = $dstOU
    $UserRecord."VDIGroup"        = $Windows10VDIGroup
    $UserRecord."GroupsToAdd"     = ""
    $UserRecord."GroupsToRemove"  = ""
    $objRecordWithUserids          = New-Object PSObject -
Property $UserRecord
    $arrTableWithUserids        += $objRecordWithUserids
    $valCounter++
```

```
   }

   If($arrTableWithUserids.Count -gt 0)
   {
      $arrTableWithUserids | Export-Csv $strFileName -
NoTypeInformation
   }
    Else
   {
     Write-Host "Something went wrong while writing the file
$strFileName. Maybe nothing to report..."
   }
}

Function Add-TrailingBackSlash
{

<#
.NOTES
===========================================================
===========================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Add-TrailingBackSlash
===========================================================
===========================================================
================
.SYNOPSIS

This function adds a trailing backslash to a string
#>

param
(
 [string] $AddBackslashTo =""
)

if($AddBackslashTo.Length -gt 0)
```

```
 {
  if(($AddBackslashTo.SubString($AddBackslashTo.Length-1,1)) -
ne "\")
  {
   $AddBackslashTo = $AddBackslashTo + "\"
  }
 }
Else
 {
  $AddBackslashTo = "\"
 }

 Return $AddBackslashTo

}

Function Copy-UserProfileFiles
{
<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Copy-UserProfileFiles
Source:
https://techblog.dorogin.com/powershell-how-to-recursively-cop
y-a-folder-structure-excluding-some-child-folders-and-files-
a1de7e70f1b
================================================================
================================================================
================
.SYNOPSIS

This function really copies the files.
#>

param
```

```
 (
  [string] $OldFilesDirectory,
  [string] $NewFilesDirectory
 )

 $arrExclude       = @("desktop.ini","$RECYCLE.BIN","Google
Chrome.lnk","thumbs.db")
 $arrExcludeMatch = @(".pst")
 New-Item -Path $NewFilesDirectory -ItemType Directory -Force
-WhatIf:(-not($ProductionRun)) | Out-Null
  Try
   {
    Get-ChildItem -Path $OldFilesDirectory -Recurse -Exclude
$arrExclude |
         where  {  $arrExcludeMatch  -eq  $null  -or
$_.FullName.Replace($OldFilesDirectory,  "")  -notmatch
$arrExcludeMatch } |
    Copy-Item -Destination {
        if ($_.PSIsContainer)
        {
                        Join-Path  $NewFilesDirectory
$_.Parent.FullName.Substring($OldFilesDirectory.length)
        }
         else
        {
                        Join-Path  $NewFilesDirectory
$_.FullName.Substring($OldFilesDirectory.length)
        }
         }  -Force  -Exclude  $arrExclude  -WhatIf:(-
not($ProductionRun))  -Confirm:$false  -Erroraction
SilentlyContinue
     Write-EntryToResultsFile -strUserid $Userid -Result
"Success"  -Action  "Copy  from  $OldFilesDirectory  to
$NewFilesDirectory."
   }
    Catch
   {
     Write-EntryToResultsFile -strUserid $Userid -Result
"Error"  -Action  "Copy  from  $OldFilesDirectory  to
$NewFilesDirectory." -Message $_.Exception.Message
    Continue
```

```
   }

}

Function Copy-UserProfile
{
<#
.NOTES
===========================================================================
===========================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Copy-UserProfile
===========================================================================
===========================================================================
================
.SYNOPSIS

This function copies the favorites and desktop from the old
profile to the new profile

#>

param
 (
  [string] $Userid="",
  [string] $OldProfilePath="",
  [string] $NewProfilePath=""
 )

 # Perform some checking
 $bolErrorsFound = $False

  $OldProfilePath = (Add-TrailingBackSlash -AddBackslashTo
$OldProfilePath) + $Userid
  $NewProfilePath = (Add-TrailingBackSlash -AddBackslashTo
$NewProfilePath) + $Userid
```

```
 if(-not(test-path($OldProfilePath)))
 {
  $bolErrorsFound = $True
  Write-EntryToResultsFile -strUserid $Userid -Result "Error"
-Action "Check directory" -Message "The directory
$OldProfilePath does not exists."
 }

 if(-not(test-path($NewProfilePath)))
 {
  $bolErrorsFound = $True
  Write-EntryToResultsFile -strUserid $Userid -Result "Error"
-Action "Check directory" -Message "The directory
$NewProfilePath does not exists."
 }

 if(test-path($NewProfilePath + "\Desktop\Oude desktop Windows
7"))
 {
  $bolErrorsFound = $True
  Write-EntryToResultsFile -strUserid $Userid -Result "Error"
-Action "Check directory" -Message "The directory
$($NewProfilePath + "\Desktop\Oude desktop Windows 7") exists.
That means that the profile has already been copied."
 }

 If($bolErrorsFound)
 {
  Return
 }

 # Copy the Favorites and desktop

   Copy-UserProfileFiles -OldFilesDirectory ($strOldFolder =
$OldProfilePath   +"\Favorites")   -NewFilesDirectory
($strNewFolder = $NewProfilePath +"\Favorites")
   Copy-UserProfileFiles -OldFilesDirectory ($strOldFolder =
$OldProfilePath  +"\Desktop")      -NewFilesDirectory
($strNewFolder = $NewProfilePath +"\Desktop\Oude desktop
Windows 7")
```

```
}

Function Add-RollBackRecord
{

<#
.NOTES
===========================================================
===========================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Add-RollBackRecord
===========================================================
===========================================================
================
.SYNOPSIS

This function adds an item to the rollback file

#>

  param
   (
    [string] $Userid,
    [string] $NewOU,
    [string] $VDIGroup,
    [string] $GroupsToAdd,
    [string] $GroupsToRemove
   )
  $RollBackRecord                        = [ordered] @{"Userid"
=     "";"NewOU"="";"VDIGroup"=     "";"GroupsToAdd"=
"";"GroupsToRemove"= ""}
  $RollBackRecord."Userid"              = $Userid
  $RollBackRecord."NewOU"               = $NewOU
  $RollBackRecord."VDIGroup"            = $VDIGroup
  $RollBackRecord."GroupsToAdd"         = $GroupsToAdd
  $RollBackRecord."GroupsToRemove"      = $GroupsToRemove
  $objRollBackRecord                     = New-Object PSObject
```

```
-Property $RollBackRecord
  $Global:arrTableWithRollBackRecords  += $objRollBackRecord
  }

#
================================================================
================================================================
================
# End function block
#
================================================================
================================================================
================

#
================================================================
================================================================
================
# Declares the variables.
# Modify $strPrefixOldGroupName for your own enfironment.
#
================================================================
================================================================
================

  $valCounter                         = 1
  $Global:arrTable                    = @()
  $Global:arrTableWithRollBackRecords = @()
  $strCurrentPath                        = Split-Path -parent
$MyInvocation.MyCommand.Definition
    $strCurrentFile                                    =
$MyInvocation.MyCommand.Name
  $strPrefixOldGroupName              = "gg_appl_"
  $strPrefixNewGroupName              = "Appl_"
         $arrGroupMustContainForFullCleanUp          =
@($strPrefixOldGroupName,"WM-Users")

#
================================================================
================================================================
================
```

```
# check the length of the current directory. Stop the script if more that 248 characters. Otherwise the results cannot be written to the given
# folder.
#
================================================================================
================================================================================

  If($strCurrentPath.Length -gt 248)
  {
    Write-Host "The current directory $strCurrentPath has a length of more than 248 characters. The script will end with exit code 999 now."
   Exit 999
  }

#
================================================================================
================================================================================
# Check if the parameter CreateFileWithUseridsInCSVFormat is used. In that case, create the file and stop.
#
================================================================================
================================================================================

  if($CreateFileWithUseridsInCSVFormat)
  {
    Create-FileWithUseridsInCSVFormat -srcOU $srcOU -dstOU $dstOU -Windows10VDIGroup $Windows10VDIGroup
   Exit 0
  }
#
================================================================================
================================================================================
# Define the CSV Import File.
#
================================================================================
```

```
================================================================
================

  if($FileWithUseridsInCSVFormat.Length -eq 0)
   {
    $strCSVFileName = $strCurrentFile -Replace ".ps1",".csv"
   }
    else
   {
    if($FileWithUseridsInCSVFormat.ToLower().IndexOf(".csv") -
eq -1)
    {
     $FileWithUseridsInCSVFormat = $FileWithUseridsInCSVFormat
+ ".csv"
    }
    $strCSVFileName = $FileWithUseridsInCSVFormat
   }

#
================================================================
================================================================
================
# Check if the string $strCSVFileName is a path. In that case,
nothing has to be done.
# In case it is not a path, then the current location should
be added.
#
================================================================
================================================================
================

  if (-not(Split-Path($strCSVFileName)))
  {
   $strCSVFileName = $strCurrentPath + "\" + $strCSVFileName
  }

#
================================================================
================================================================
================
# In case of an automated migration check if the string
```

```
$FileForAutomaticMigration is a path. In that case, nothing
has to be done.
# In case it is not a path, then the current location should
be added.
# If the file exists then read all the content, otherwise
quit.
#
================================================================
================================================================
================

  if($FileForAutomaticMigration.Length -gt 0)
   {
    if (-not(Split-Path($FileForAutomaticMigration)))
    {
      $FileForAutomaticMigration = $strCurrentPath + "\" +
$FileForAutomaticMigration
    }
    if(Test-Path($FileForAutomaticMigration))
     {
        $arrGroupNamesForAutomaticMigration = @(Import-Csv
$FileForAutomaticMigration)
      if ($arrGroupNamesForAutomaticMigration.Count -eq 0)
      {
      Write-Host "The file $FileForAutomaticMigration seems to
be empty..."
      Exit 1
      }
     }
      else
     {
      Write-Host "The file $FileForAutomaticMigration does not
exists. Thus quitting."
      Exit 1
     }
   }

#
================================================================
================================================================
================
```

```
# Define the log file. This log file contains all the results.
#
================================================================
================================================================
================

  $strLastPartOfFileName = " (" + (Get-Date).ToString('G') +
").csv"
   $strLastPartOfFileName = $strLastPartOfFileName -replace
":","-"
   $strLastPartOfFileName = $strLastPartOfFileName -replace
"/","-"
  If(-not($ProductionRun))
  {
    $strLastPartOfFileName = " (RUNNING IN TEST MODE)" +
$strLastPartOfFileName
  }

  $strCSVLogFileSucces   = $strCSVFileName -Replace ".csv",
$strLastPartOfFileName

  $strPathName          = (Split-Path $strCSVLogFileSucces) +
"\"
       $strFileName                                        =
$strCSVLogFileSucces.Substring($strPathName.Length,($strCSVLog
FileSucces.Length - $strPathName.Length))

  $strCSVLogFileSucces   = $strPathName + $LogFilePrefix +
$strFileName
   $strCSVRollBackFile     = $strPathName + "RBF_" +
$strFileName

  $valFileLengthLogFile = $strCSVLogFileSucces.Length

  if($valFileLengthLogFile -gt 260)
  {
   Write-Host "The file name $strCSVLogFileSucces is too long.
The maximum file length is 260 characters. This one is
$valFileLengthLogFile long.`n No log file is generated, and
therefore, this application will quit."
   Exit 2
```

```
  }

#
================================================================
================================================================
================
# Read the CSV file.
#
================================================================
================================================================
================
  If(Test-Path $strCSVFileName)
  {
   $arrUserids = @(Import-Csv $strCSVFileName)
  }
   Else
  {
    Write-Host "The import file $strCSVFileName does not
exists."
   Exit 1
  }
#
================================================================
================================================================
================
# Find all the arguments and puthem in the log file
# Source: https://ss64.com/ps/psboundparameters.html
#
================================================================
================================================================
================

  Write-EntryToResultsFile -strUserid "" -Result "Information"
-Action "Used script" -Message $strCurrentPath + "\" +
$strCurrentFile

  foreach($boundparam in $PSBoundParameters.GetEnumerator())
  {
     Write-EntryToResultsFile -strUserid "" -Result
"Information" -Action "Key: $($boundparam.Key)" -Message
"Value: $($boundparam.Value)"
```

```
  }
#
================================================================
================================================================
================
# Process the users in the CSV file.
#
================================================================
================================================================
================

  Clear-Host

  $strActivity                          = "Modifying users in
Active Directory."

  If(-not($ProductionRun))
  {
  $strActivity = $strActivity + " (RUNNING IN TEST MODE)"
  }

  ForEach($objUser in $arrUserids)
  {
   $strUserid = $objUser.Userid
    Write-Progress -Activity $strActivity -Status "Processing
user  $strUserid"  -PercentComplete  ($valCounter  /
$arrUserids.Count * 100)

   if($CreateRollBackFile)
   {
    $Global:strForRollBackGroupsToAdd    = ""
    $Global:strForRollBackGroupsToRemove = ""
     $strUserDetails                          = Get-ADUser
$strUserid -Properties cn, DistinguishedName
     $strCurrentUserOU                         = "OU=" +
($strUserDetails.DistinguishedName -split "=",3)[-1]
   }

   Try
   {
    $strNewOU          = $objUser.NewOU
```

```
    $strVDIGroup       = $objUser.VDIGroup
    $arrGroupsToAdd    = $objUser.GroupsToAdd.Split(",")
    $arrGroupsToRemove = $objUser.GroupsToRemove.Split(",")
   }
    Catch
   {
     Write-EntryToResultsFile -strUserid $strUserid -Result
"Error" -Message "There is something wrong with the CSV
filename $strCSVFileName while processing $strUserid."
    Export-ResultsLogFileToCSV
    Exit 1
   }
   # Copy Profile
   if($copyProfile)
   {
      Copy-UserProfile -Userid $strUserid -OldProfilePath
$ProfilePathFrom -NewProfilePath $ProfilePathTo
   }

   if(-not $copyProfile)
   {

    # Clear the profile path
    If($ClearProfilePath)
     {
         Remove-ProfilePathFromUserProfileInAD -strUserid
$strUserid
     }
    # Clear the homedrive and home directory
    If($ClearHomeFolder)
     {
        Remove-HomeFolderPathFromUserProfileInAD -strUserid
$strUserid
     }
    # Move the user to another OU:
       Move-ADUserToOtherOU -strUserid $strUserid -
strDestinationOU $strNewOU
    # Add the user to the new VDI group:
     Add-ADMemberToGroup -strUserid $strUserid -strADGroupName
$strVDIGroup
    # Add the user to various groups:
```

```
    ForEach($objGroupToAdd in $arrGroupsToAdd)
    {
     Add-ADMemberToGroup -strUserid $strUserid -strADGroupName
$objGroupToAdd
    }
    # Remove the user to various groups:
    ForEach($objGroupToRemove in $arrGroupsToRemove)
    {
       Remove-ADMemberFromGroup -strUserid $strUserid -
strADGroupName $objGroupToRemove
    }
   }

   # Automatic migration

    if($FileForAutomaticMigration.Length -gt 0 -and -not
$copyProfile)
   {
      $arrGroups = @(Get-ADUser $strUserid -Properties
MemberOf).MemberOf
    forEach($objGroup in $arrGroups)
    {
     $strGroupName = (Get-ADGroup $objGroup).Name
                                                          if
($strGroupName.ToLower().IndexOf($strPrefixOldGroupName)  -eq
0)
     {
        Migrate-ADUserToNewGroup -strUserid $strUserid -
strADGroupName $strGroupName
     }
    }
   }

   # Full Clean Up
   if($FullCleanUp -and -not $copyProfile)
   {
     Remove-UserFromMultipleGroups  -arrGroupsToSearchFor
$arrGroupMustContainForFullCleanUp -strUserid $strUserid
   }

   # Remove old Citrix groups
```

```
   if($RemoveOldCitrixGroups -and -not $copyProfile)
   {
                                    $arrCitrixOU          =
@("OU=Citrix,OU=OU1,DC=testdomain,DC=local,DC=lan")

   ForEach($objCitrixOU in $arrCitrixOU)
     {
       $arrWithCitrixGroups = get-adgroup -filter "*" -
SearchBase $objCitrixOU | Where-Object Name -Match "Citrix
VDI*"
     ForEach ($objWithCitrixGroups in $arrWithCitrixGroups)
      {
        Remove-UserFromMultipleGroups -arrGroupsToSearchFor
$objWithCitrixGroups.Name -strUserid $strUserid
      }
     }

       Remove-UserFromMultipleGroups -arrGroupsToSearchFor
@("CitrixTestUsers","CitrixProductionUsers")    -strUserid
$strUserid
   }

   # Perform the Direct Group Inventory
   if($RunDirectGroupInventory -and -not $ProductionRun -and -
not $copyProfile)
   {
    Show-DirectGroupsInResultsFile -strUserid $strUserid
   }

   # Perform the Indirect Group Inventory
   if($RunIndirectGroupInventory -and -not $ProductionRun -and
-not $copyProfile)
   {
    Show-IndirectGroupsInResultsFile -strUserid $strUserid
   }

   if($CreateRollBackFile)
   {
       Add-RollBackRecord -Userid $strUserid -NewOU
$strCurrentUserOU    -VDIGroup    ""    -GroupsToAdd
$Global:strForRollBackGroupsToAdd      -GroupsToRemove
```

```
$Global:strForRollBackGroupsToRemove
   }

   $valCounter++
  }
   if($ProductionRun -and ($RunDirectGroupInventory -or
$RunIndirectGroupInventory) -and -not $copyProfile)
  {
   $valCounter = 1
   $maxCounter = 30
         For($valCounter = 1; $valCounter -le
$maxCounter;$valCounter++)
    {
     Write-Progress -Activity "Processing changes in Active
Directory" -Status "Waiting $valCounter of $maxCounter
seconds." -PercentComplete ($valCounter / $maxCounter * 100)
     Sleep 1
    }

                                                              #
================================================================
================================================================
================
   # Inventory the indirect user groups the user belongs to.
   # This can only be done after updating Active Directory,
otherwise incorrect results are shown.
                                                              #
================================================================
================================================================
================

     $strActivity                          = "Inventory the
indirect and / or direct groups the user belongs to."

     If(-not($ProductionRun))
      {
       $strActivity = $strActivity + " (RUNNING IN TEST MODE)"
      }

     $valCounter = 1
     ForEach($objUser in $arrUserids)
```

```
      {
       $strUserid = $objUser.Userid
          Write-Progress -Activity $strActivity -Status
"Processing user $strUserid" -PercentComplete ($valCounter /
$arrUserids.Count * 100)
       if ($RunDirectGroupInventory)
        {
         Show-DirectGroupsInResultsFile -strUserid $strUserid
        }
       if ($RunIndirectGroupInventory)
        {
            Show-IndirectGroupsInResultsFile -strUserid
$strUserid
        }
       $valCounter++
      }
  }

#
================================================================
================================================================
================
# Write the results to the csv file.
#
================================================================
================================================================
================
  Export-ResultsLogFileToCSV
#
================================================================
================================================================
================
# Write the Rollback file to a csv file.
#
================================================================
================================================================
================

  if($CreateRollBackFile)
  {
   Export-RollBackFileToCSV
```

```
}
```

Current version: Migrate users (v10)

Any feedback to improve this script is appreciated. You can download the previous versions here:

1. Migrate users (v01)
2. Migrate users (v02)
3. Migrate users (v03)

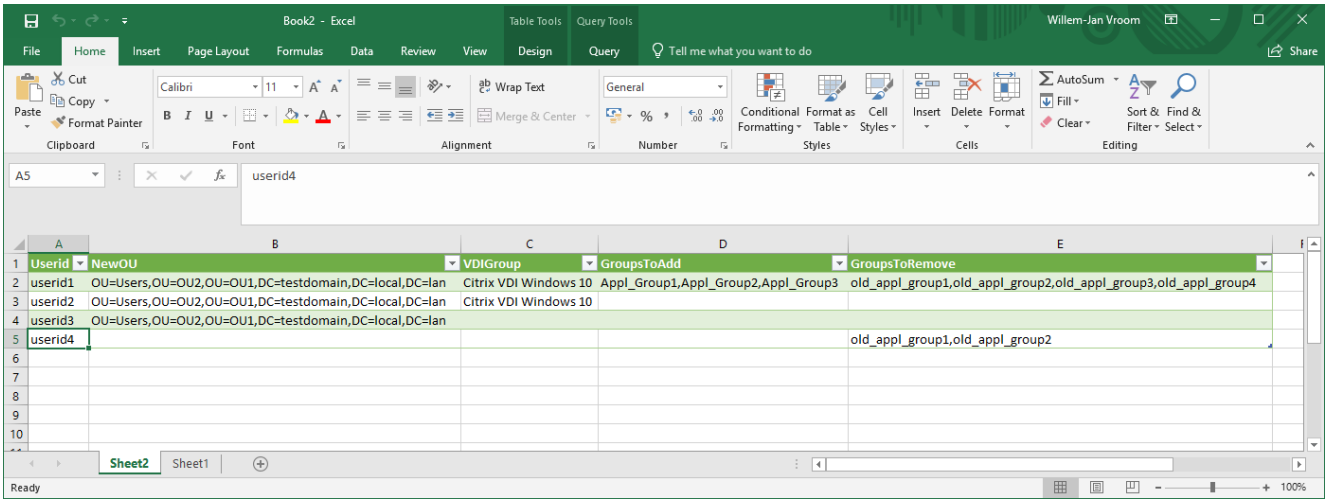
Migrate users from Windows 7 to Windows 10 automatically

For a migration from Windows 7 to Windows 10 it is needed to do the following with the users' account:

- Move it to a different OU
- Add it to a new VDI group
- Add it to the new application groups
- Remove it from the previous application groups
- Clear the profile path, if needed.

To do this, I created a Powershell script.

The users who need to be migrated from Windows 7 to Windows 10 are in a csv file:



Userid	NewOU	VDIGroup	GroupsToAdd	GroupsToRemove
userid1	OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan	Citrix VDI Windows 10	Appl_Group1,Appl_Group2,Appl_Group3	old_appl_group1,old_appl_group2,old_appl_group3,old_appl_group4
userid2	OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan	Citrix VDI Windows 10		
userid3	OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan			
userid4				old_appl_group1,old_appl_group2

- The column **Userid** contains the userid
- The column **NewOU** contains the OU where the user should be moved to. This column can be empty.
- The colum **VDIGroup** contains the new VDI group. This column can be empty.
- The colum **GroupsToAdd** contains the groups the user should be added to. Split the groups with a comma. This column can be empty.
- The colum **GroupsToRemove** contains the groups the user should be removed from. Split the groups with a comma. This column can be empty.

An example:

```
"Userid","NewOU","VDIGroup","GroupsToAdd","GroupsToRemove"
"userid1","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan","Citrix VDI Windows 10","Appl_Group1,Appl_Group2,Appl_Group3","old_appl_group1,old_appl_group2,old_appl_group3,old_appl_group4"
"userid2","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan","Citrix VDI Windows 10","",""
"userid3","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=lan","","","",""
"userid4","","","","old_appl_group1,old_appl_group2"
```

The are some parameters:

- **FileWithUseridsInCSVFormat**: CSV Filename that contains

all the userids that should be migrated. If not mentioned than the script name is used.
- **LogFilePrefix**: The name the logfile starts with. So the logfiles are grouped together. Default = ZZZ-Logfile_
- **ProductionRun**: Use the swith ProductionRun to modify. If not specified, the script is run in test mode.
- **FullCleanUp**: Use the swith FullCleanUp to remove all unneeded groups.
- **ClearProfilePath**: Use the swith ClearProfilePath to clear the profile path.

The script:

```
<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Filename:         Migrate users (v03).ps1
================================================================
================================================================
================
.DESCRIPTION:

This script prepares the users for the migration from Windows
7 to Windows 10.

.USAGE:

Create a CSV file with the following layout:

"Userid","NewOU","VDIGroup","GroupsToAdd","GroupsToRemove"
"userid1","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=la
n","Citrix                    VDI                    Windows
10","Appl_Group1,Appl_Group2,Appl_Group3","old_appl_group1,old
_appl_group2,old_appl_group3,old_appl_group4"
```

```
"userid2","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=la
n","Citrix VDI Windows 10","",""
"userid3","OU=Users,OU=OU2,OU=OU1,DC=testdomain,DC=local,DC=la
n","","",""
"userid4","","","","old_appl_group1,old_appl_group2"

.PARAMETERS:

-FileWithUseridsInCSVFormat:
The filename with the users to be modified.

-LogFilePrefix
The name the logfile starts with. So the logfiles are grouped
together. Default = ZZZ-Logfile_

-ProductionRun
If the switch is used then ActiveDirectory is modified. If not
used then a test run is done, and Active Directory is not
modified.

-FullCleanUp
If this switch is used then the following groups will be
removed from the users' account:
 * All gg_appl groups
 * WM-Users

 -ClearProfilePath
 If this switch is used then the profile path will be cleared.
.VERSION HISTORY:
 v0.1:
   * Initial version.

 v0.2:
   * The logfilename has been changed.

 V0.3:
   * The logfile mentions 'RUNNING IN TEST MODE' in case the
swith -ProductionRun is not used.
#>

param
```

```
(
[Parameter(HelpMessage="CSV Filename that contains all the userids that should be migrated. Default = the script name, with the csv extension.")]
[String] $FileWithUseridsInCSVFormat="",

[Parameter(HelpMessage="The name the logfile starts with. So the logfiles are grouped together. Default = ZZZ-Logfile_")]
[String] $LogFilePrefix = "ZZZ-Logfile_",

[Parameter(HelpMessage="Use the swith ProductionRun to modify. If not specified, the script is run in test mode.")]
[Switch] $ProductionRun,

[Parameter(HelpMessage="Use the swith FullCleanUp to remove all unneeded groups.")]
[Switch] $FullCleanUp,

[Parameter(HelpMessage="Use the swith ClearProfilePath to clear the profile path.")]
[Switch] $ClearProfilePath
)

#
================================================================
================================================================
================
# Function block
#
================================================================
================================================================
================

Function Write-EntryToResultsFile
{

<#
.NOTES
================================================================
================================================================
================
```

```
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Write-EntryToResultsFile
=============================================================
=============================================================
================
.DESCRIPTION:

This function adds the success or failure information to the
array that contains the log
information.

#>
param
 (
  $strUserid,
  $ErrorMessage = "",
  $Action       = ""
 )
 $Record             = [ordered] @{"Username" = "";"Action"=
"";"Testmode"="";"Error"= ""}
 $Record."Username" = $strUserid
 $Record."Action"   = $Action
 $Record."Testmode" = -not($ProductionRun)
 $Record."Error"    = $ErrorMessage
 $objRecord         = New-Object PSObject -Property $Record
 $Global:arrTable   += $objRecord
}

Function Remove-ProfilePathFromUserProfileInAD
{

<#
.NOTES
=============================================================
=============================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
```

```
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Remove-ProfilePathFromUserProfileInAD
=============================================================
=============================================================
================
.DESCRIPTION:

This function clears the ProfilePath from AD.

#>

  param
   (
    $strUserid
   )
   Try
      {
                $strUserDN  =  (Get-ADUser  -Identity
$strUserid).distinguishedName
       Set-ADUser -Identity $strUserDN -Clear profilePath -
WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Clear profile path"
      }
       Catch
      {
         Write-EntryToResultsFile -strUserid $strUserid -
ErrorMessage $_.Exception.Message -Action "Clear profile path"
       Continue
      }
}

Function Move-ADUserToOtherOU
{

<#
.NOTES
=============================================================
=============================================================
================
```

```
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Move-ADUserToOtherOU
=============================================================
=============================================================
================
.DESCRIPTION:

This function moves the user to another OU.

#>

  param
   (
    $strUserid,
    $strDestinationOU
   )
   Try
     {
      if($strDestinationOU.Length -gt 0)
      {
                $strUserDN  =  (Get-ADUser  -Identity
$strUserid).distinguishedName
         Move-ADObject -Identity $strUserDN -TargetPath
$strDestinationOU   -WhatIf:(-not($ProductionRun))   -
Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Move AD User to OU $strDestinationOU."
      }
     }
      Catch
     {
        Write-EntryToResultsFile -strUserid $strUserid -
ErrorMessage $_.Exception.Message -Action "Move AD User to OU
$strDestinationOU."
      Continue
     }
}
```

```
Function Add-ADMemberToGroup
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Add-ADMemberToGroup
================================================================
================================================================
================
.DESCRIPTION:

This function adds a user to an AD group.

#>

  param
   (
    $strUserid,
    $strADGroupName
   )
   Try
     {
      if($strADGroupName.Length -gt 0)
      {
                $strUserDN  =  (Get-ADUser  -Identity
$strUserid).distinguishedName
        Add-ADGroupMember -Identity $strADGroupName -Members
$strUserDN -WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Add AD group $strADGroupName to user."
      }
     }
      Catch
     {
```

```
        Write-EntryToResultsFile -strUserid $strUserid -
ErrorMessage $_.Exception.Message -Action "Add AD group
$strADGroupName to user."
      Continue
     }
}

Function Remove-ADMemberFromGroup
{

<#
.NOTES
================================================================
================================================================
================
Created with:     Windows PowerShell ISE
Created on:       06-September-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Remove-ADMemberFromGroup
================================================================
================================================================
================
.DESCRIPTION:

This function removes a user from an AD group.

#>

  param
   (
    $strUserid,
    $strADGroupName
   )
   Try
     {
      if($strADGroupName.Length -gt 0)
      {
                $strUserDN  =  (Get-ADUser  -Identity
$strUserid).distinguishedName
       Remove-ADGroupMember -Identity $strADGroupName -Members
```

```
$strUserDN -WhatIf:(-not($ProductionRun)) -Confirm:$false
       Write-EntryToResultsFile -strUserid $strUserid -Action
"Remove AD group $strADGroupName from user."
      }
     }
      Catch
     {
        Write-EntryToResultsFile -strUserid $strUserid -
ErrorMessage $_.Exception.Message -Action "Remove AD group
$strADGroupName from user."
      Continue
     }
}

#
================================================================
================================================================
================
# End function block
#
================================================================
================================================================
================

#
================================================================
================================================================
================
# Declares the variables.
#
================================================================
================================================================
================

  $valCounter                          = 1
  $Global:arrTable                     = @()
  $strActivity                            = "Modifying users in
Active Directory."
  $arrGroupMustContainForFullCleanUp = @("gg_appl","WM-Users")
  $strCurrentPath                        = Split-Path -parent
$MyInvocation.MyCommand.Definition
```

```
    $strCurrentFile                                                 =
$MyInvocation.MyCommand.Name


#
================================================================
================================================================
================
# Define the CSV Import File.
#
================================================================
================================================================
================

  if($FileWithUseridsInCSVFormat.Length -eq 0)
   {
    $strCSVFileName = $strCurrentFile -Replace ".ps1",".csv"
   }
    else
   {
    if($FileWithUseridsInCSVFormat.ToLower().IndexOf(".csv") -
eq -1)
    {
     $FileWithUseridsInCSVFormat = $FileWithUseridsInCSVFormat
+ ".csv"
    }
    $strCSVFileName = $FileWithUseridsInCSVFormat
   }

#
================================================================
================================================================
================
# Check if the string $strCSVFileName is a path. In that case,
nothing has to be done.
# In case it is not a path, then the current location should
be added.
#
================================================================
================================================================
================
```

```
  if (-not(Split-Path($strCSVFileName)))
  {
   $strCSVFileName = $strCurrentPath + "\" + $strCSVFileName
  }

#
================================================================
================================================================
================
# Define the log file. This log file contains all the results.
#
================================================================
================================================================
================

  $strLastPartOfFileName = " (" + (Get-Date).ToString('F') +
").csv"
   $strLastPartOfFileName = $strLastPartOfFileName -replace
":","-"
  If(-not($ProductionRun))
  {
   $strLastPartOfFileName = " (RUNNING IN TEST MODE)" +
$strLastPartOfFileName
  }

  $strCSVLogFileSucces    = $strCSVFileName -Replace ".csv",
$strLastPartOfFileName

  $strPathName           = (Split-Path $strCSVLogFileSucces) +
"\"
       $strFileName                                          =
$strCSVLogFileSucces.Substring($strPathName.Length,($strCSVLog
FileSucces.Length - $strPathName.Length))

  $strCSVLogFileSucces    = $strPathName + $LogFilePrefix +
$strFileName
#
================================================================
================================================================
================
# Read the CSV file.
```

```
#
================================================================
================================================================
================
  if(Test-Path $strCSVFileName)
  {
   $arrUserids = @(Import-Csv $strCSVFileName)
  }
   Else
  {
    Write-Host "The import file $strCSVFileName does not
exists."
   Exit 1
  }
#
================================================================
================================================================
================
# Process the users in the CSV file.
#
================================================================
================================================================
================

  Clear-Host

  If(-not($ProductionRun))
  {
  $strActivity = $strActivity + " (RUNNING IN TEST MODE)"
  }

  ForEach($objUser in $arrUserids)
  {
   $strUserid = $objUser.Userid
    Write-Progress -Activity $strActivity -Status "Processing
user  $strUserid"  -PercentComplete  ($valCounter  /
$arrUserids.Count * 100)
   Try
   {
    $strNewOU          = $objUser.NewOU
    $strVDIGroup       = $objUser.VDIGroup
```

```
    $arrGroupsToAdd    = $objUser.GroupsToAdd.Split(",")
    $arrGroupsToRemove = $objUser.GroupsToRemove.Split(",")
   }
    Catch
   {
    Write-Host "There is something wrong with the CSV filename
$strCSVFileName while processing $strUserid."
    Exit 1
   }
   # Clear the profile path
   If($ClearProfilePath)
   {
       Remove-ProfilePathFromUserProfileInAD  -strUserid
$strUserid
   }
   # Move the user to another OU:
      Move-ADUserToOtherOU  -strUserid  $strUserid  -
strDestinationOU $strNewOU
   # Add the user to the new VDI group:
    Add-ADMemberToGroup -strUserid $strUserid -strADGroupName
$strVDIGroup

   # Add the user to various groups:
   ForEach($objGroupToAdd in $arrGroupsToAdd)
   {
    Add-ADMemberToGroup -strUserid $strUserid -strADGroupName
$objGroupToAdd
   }
   # Remove the user to various groups:
   ForEach($objGroupToRemove in $arrGroupsToRemove)
   {
      Remove-ADMemberFromGroup  -strUserid $strUserid  -
strADGroupName $objGroupToRemove
   }
   # Full Clean Up
   if($FullCleanUp)
   {
      $arrGroups  =  @(Get-ADUser  $strUserid  -Properties
MemberOf).MemberOf
    forEach($objGroup in $arrGroups)
    {
```

```
    $strGroup=(Get-ADGroup $objGroup).Name
                    forEach($objGroupMustContain    in
$arrGroupMustContainForFullCleanUp)
     {
                                $strGroupMustContain    =
$objGroupMustContain.ToString().ToLower()
      if($strGroup.ToLower().IndexOf($strGroupMustContain) -eq
0)
       {
          Remove-ADMemberFromGroup -strUserid $strUserid -
strADGroupName $strGroup
       }
     }
    }
   }

   $valCounter++
   Sleep 2
  }

  Sleep 2

#
================================================================
================================================================
================
# Write the results to the csv file.
#
================================================================
================================================================
================
  If($Global:arrTable.Count -gt 0)
  {
     $Global:arrTable | Export-Csv $strCSVLogFileSucces -
NoTypeInformation
  }
   Else
  {
    Write-Host "Something went wrong while writing the logfile
$strCSVLogFileSucces. Maybe nothing to report..."
  }
```

Any feedback to improve this script is appreciated. You can download the scripts here:

1. Migrate users (v01)
2. Migrate users (v02)
3. Migrate users (v03)

Inventory the directory access rights on file servers

During the migration, it was needed to inventory the directory access on several file servers. So we could easily monitor the access rights on several directories on the file servers. Some applications are started from a UNC path. So we could add the 'old' and the 'new' group and check if that has been done properly.

For testing I created the following structure:

```
\\DEMOATS-SCCM\DEMO.
\---share1
    +---Appl1
    +---Appl2
    |   \---Test
    +---Appl3
    |   +---Sub1
    |   \---Sub2
    |       \---SubSub1
    \---Appl4
```

I had some challenges:

- Make a difference between inherited and non-inherited rights. I only want to see the differences. But that can

be changed with the parameter showInherited
- There are some file servers where all the shares (from the root) have to be inventoried. \\server\share did not work, as the rights where inherited. And the script did not see that properly. So I had to inventory the 'root' shares on the server. And go through all the directories. I found the code on StackOverflow.
- And a lot of testing.

The script that I created:

```
<#
.NOTES
================================================================
================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Filename:         Inventory Permissions on Shares (v02).ps1
================================================================
================================
.DESCRIPTION:

This script writes the directory permissions of the given
shares to a CSV file.

.USAGE:

1.
Run an inventory on the shares
\\server\share,\\server2\share1 and all shares on \\server3\
with the default setting of a search level of 1 and only show
the directories that are not
inherited for the AD group 'Users':
.\"Inventory Permissions on Shares (v02).ps1" -ShareList
\\server\share,\\server2\share1,\\server3\

2.
Run an inventory on the share \\server\share with an search
```

```
level of 10 for all the 'Appl'
groups:
.\"Inventory Permissions on Shares (v02).ps1" -ShareList
\\server\share    -NumberOfLevelsToSearch    10    -
GroupNameToSearchFor "Appl"

3.
Run a complete inventory for one server for all groups:
.\"Inventory Permissions on Shares (v02).ps1" -ShareList
\\server\ -NumberOfLevelsToSearch 10 -GroupNameToSearchFor ""
-showInherited

.VERSION HISTORY:
 v0.1:
   * Initial version.

 v.0.2:
   * Option -Outputfile has been added.
   * Added help text by the options.

 v.0.3:
   * The parameter showInherited has become a switch.

#>

param
(
[Parameter(Mandatory=$true,HelpMessage="Please  mention  the
shares you want to inventory regarding the permissions. One
name each line.")]
[String[]] $ShareList,

[Parameter(HelpMessage="Give a part of the group name to
search for. Leave empty for all groups. Default = Users")]
$GroupNameToSearchFor    = "Users",

[Parameter(HelpMessage="Give the search level. Default = 1")]
$NumberOfLevelsToSearch  = 1,

[Parameter(HelpMessage="Show  inherited  directories,  if
specified.")]
```

```
[switch]$showInherited,

[Parameter(HelpMessage="Mention the output file. Default is
the script name, with csv as the extension.")]
$OutputFile             = ""
)

#
================================================================
================================
# Function block
#
================================================================
================================

Function Get-NetShares
{

<#
.NOTES
================================================================
================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created                                                    by:
https://stackoverflow.com/users/2102693/bill-stewart
Organization:
Functionname:     Get-NetShares
================================================================
================================
.DESCRIPTION:

This function finds all the shares that are on a server.

I have found this script here:
https://stackoverflow.com/questions/45089582/using-get-childit
em-at-root-of-unc-path-servername
(C) by https://stackoverflow.com/users/2102693/bill-stewart

#>
```

```
param
(
  [String] $ComputerName = "."
)

Add-Type @"
using System;
using System.Runtime.InteropServices;
using System.Text;
[StructLayout(LayoutKind.Sequential,    CharSet    =
CharSet.Unicode)]
public struct SHARE_INFO_1
{
  [MarshalAs(UnmanagedType.LPWStr)]
  public string shi1_netname;
  public uint shi1_type;
  [MarshalAs(UnmanagedType.LPWStr)]
  public string shi1_remark;
}
public static class NetApi32
{
  [DllImport("netapi32.dll", SetLastError = true)]
  public static extern int NetApiBufferFree(IntPtr Buffer);
   [DllImport("netapi32.dll",  CharSet  =  CharSet.Unicode,
SetLastError = true)]
  public static extern int NetShareEnum(
    StringBuilder servername,
    int level,
    ref IntPtr bufptr,
    uint prefmaxlen,
    ref int entriesread,
    ref int totalentries,
    ref int resume_handle);
}
"@

$pBuffer = [IntPtr]::Zero
$entriesRead = $totalEntries = $resumeHandle = 0
$result = [NetApi32]::NetShareEnum(
  $ComputerName,        # servername
  1,                    # level
```

```
  [Ref] $pBuffer,       # bufptr
  [UInt32]::MaxValue,   # prefmaxlen
  [Ref] $entriesRead,   # entriesread
  [Ref] $totalEntries,  # totalentries
  [Ref] $resumeHandle   # resumehandle
)
if ( ($result -eq 0) -and ($pBuffer -ne [IntPtr]::Zero) -and
($entriesRead -eq $totalEntries) ) {
  $offset = $pBuffer.ToInt64()
  for ( $i = 0; $i -lt $totalEntries; $i++ ) {
    $pEntry = New-Object IntPtr($offset)
                                        $shareInfo         =
[Runtime.InteropServices.Marshal]::PtrToStructure($pEntry,
[Type] [SHARE_INFO_1])
    $shareInfo
                                          $offset          +=
[Runtime.InteropServices.Marshal]::SizeOf($shareInfo)
  }
  [Void] [NetApi32]::NetApiBufferFree($pBuffer)
}
if ( $result -ne 0 ) {
           Write-Error     -Exception     (New-Object
ComponentModel.Win32Exception($result))
}
}

Function Add-EntryToReport
{

<#
.NOTES
================================================================
================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Add-EntryToReport
================================================================
================================
.DESCRIPTION:
```

```
This function adds an entry to the report. When the shares
have been searched, the report is
exported to a CSV file.

#>

param
 (
  $FolderNameToAdd,
  $ErrorMessage = "",
  $ADGroup      = "",
  $Permissions  = "",
  $Inherited    = ""
 )
   $Record = [ordered] @{"FolderName" = "";"AD Group" =
"";"Permissions" = "";"Inherited" = "";"Error" = ""}
  $Record."FolderName"  = $FolderNameToAdd
  $Record."Error"       = $ErrorMessage
  $Record."AD Group"    = $ADGroup
  $Record."Permissions" = $Permissions
  $Record."Inherited"   = $Inherited
   $Global:Report += New-Object -TypeName PSObject -Property
$Record
}


Function Search-InTheFolder
{

<#
.NOTES
================================================================
================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Search-InTheFolder
================================================================
================================
.DESCRIPTION:
```

```
This function goes through all the folders in the given
location.

The variable $numLevels gives the number of levels that the
search goes. The less the number,
the quicklier the script is.

#>

param
(
  $RootOfTheShare,
  $numLevels = 1
)

$valNumberOfDirectories = 0
$valCounterOfDirectores = 0

Try
  {
$FolderPath = Get-ChildItem -Path $RootOfTheShare -Directory -
Recurse -Force -Depth $numLevels -ErrorAction SilentlyContinue
Write-Progress -Activity "Counting the number of folders in
the share." -Id 2 -ParentId 1
Foreach ($Folder in $FolderPath)
     {
      $valNumberOfDirectories++
     }
   Foreach ($Folder in $FolderPath)
     {
      $FolderFullName = $Folder.FullName
      Write-Progress -Activity "Going through all the shares."
-Status "Processing share $FolderFullName." -Id 2 -
PercentComplete      ($valCounterOfDirectores      /
$valNumberOfDirectories * 100) -ParentId 1
          Get-FolderRights -FolderNameToInvestigate
$Folder.FullName
      $valCounterOfDirectores++
     }
  }
  Catch
```

```
  {
   Add-EntryToReport  -FolderNameToAdd  $RootOfTheShare  -
ErrorMessage $_.Exception.Message
  }

}


Function Get-FolderRights
{

<#
.NOTES
================================================================
================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Get-FolderRights
================================================================
================================
.DESCRIPTION:

This function puts the access information in an array.
If the parameter GroupNameToSearchFor is empty or "" then all
the groups are shown.

#>
Param($FolderNameToInvestigate)
$GroupNameToSearchFor = $GroupNameToSearchFor.ToLower()
Try
    {
    $Acl = Get-Acl -Path $FolderNameToInvestigate -ErrorAction
SilentlyContinue
    foreach ($Access in $acl.Access)
        {
              $Group=$Access.IdentityReference
              if ($GroupNameToSearchFor.Length -ge 1)
               {
                                                  $Position   =
```

```
$Group.ToString().ToLower().IndexOf($GroupNameToSearchFor)
                }
                 Else
                {
                 $Position = 2
                }
               if($Position -ge 1)
               {
                [bool]$blnIsInheriated = $Access.IsInherited
                if([bool]$showInherited)
                 {
                      Add-EntryToReport -FolderNameToAdd
$FolderNameToInvestigate -ADGroup $Access.IdentityReference -
Permissions    $Access.FileSystemRights    -Inherited
[bool]$blnIsInheriated
                 }
                   if((-not[bool]$showInherited) -and (-
not[bool]$blnIsInheriated))
                 {
                      Add-EntryToReport -FolderNameToAdd
$FolderNameToInvestigate -ADGroup $Access.IdentityReference -
Permissions    $Access.FileSystemRights    -Inherited
[bool]$blnIsInheriated
                 }
               }
        }
        }
        Catch
        {
                    Add-EntryToReport  -FolderNameToAdd
$FolderNameToInvestigate -ErrorMessage $_.Exception.Message
        Continue
        }
}

#
================================================================
================================
# End function block
#
================================================================
```

```
================================

#
================================================================
================================
# Define the CSV Export File.
#
================================================================
================================

  $currentPath                        = Split-Path -parent
$MyInvocation.MyCommand.Definition
    $strCurrentFile                                       =
$MyInvocation.MyCommand.Name
  if($OutputFile.Length -eq 0)
   {
    $strCSVFileName                 = $strCurrentFile -Replace
".ps1",".csv"
   }
    else
   {
    if($OutputFile.ToLower().IndexOf(".csv") -eq -1)
    {
     $OutputFile = $OutputFile + ".csv"
    }
    $strCSVFileName = $OutputFile
   }

#
================================================================
================================
# Check if the string $strCSVFileName is a path. In that case,
nothing has to be done.
# In case it is not a path, then the current location should
be added.
#
================================================================
================================


  if (Split-Path($strCSVFileName))
```

```
  {
   $CSVExportFile = $strCSVFileName
  }
   else
  {
   $CSVExportFile = $currentPath + "\" + $strCSVFileName
  }

#
================================================================
==================================
# Create the folder as a part of $CSVExportFile if not exists.
#
================================================================
==================================

  $PathFromCSVExportFile = Split-Path $CSVExportFile
  if(-not(Test-Path(Split-Path $PathFrom$CSVExportFile)))
  {
   New-Item -Path $PathFromCSVExportFile -ItemType Directory
  }

#
================================================================
==================================
# Define variables.
#
================================================================
==================================

  $arrShares                        = @($ShareList)
  $Global:Report                    = @()
#
================================================================
==================================
# Start the job.
#
================================================================
==================================

  Clear-Host
```

```
  Try
     {
      Import-Module ActiveDirectory
     }
     Catch
     {
      Write-Host The module ActiveDirectory could not be
loaded.
     Exit 1
     }
  Write-Host (Get-Date).ToString('T') " Starting..."

#
================================================================
================================
# Deletes the CSV file if exists.
#
================================================================
================================

  If(Test-Path $CSVExportFile)
  {
  Remove-Item $CSVExportFile
  }

#
================================================================
================================
# Go through all the shares as defined in the array $arrShares
#
================================================================
================================
  $valCounter       = 1
  $valNumerOfShares = $arrShares.Count
  ForEach ($shareName in $arrShares)
  {
   Write-Progress -Id 1 -Activity "Going through the shares" -
Status  "Checking  share  $shareName  ($valCounter  of
$valNumerOfShares)"  -PercentComplete  ($valCounter  /
$valNumerOfShares*100)
                                $LastCharacter          =
```

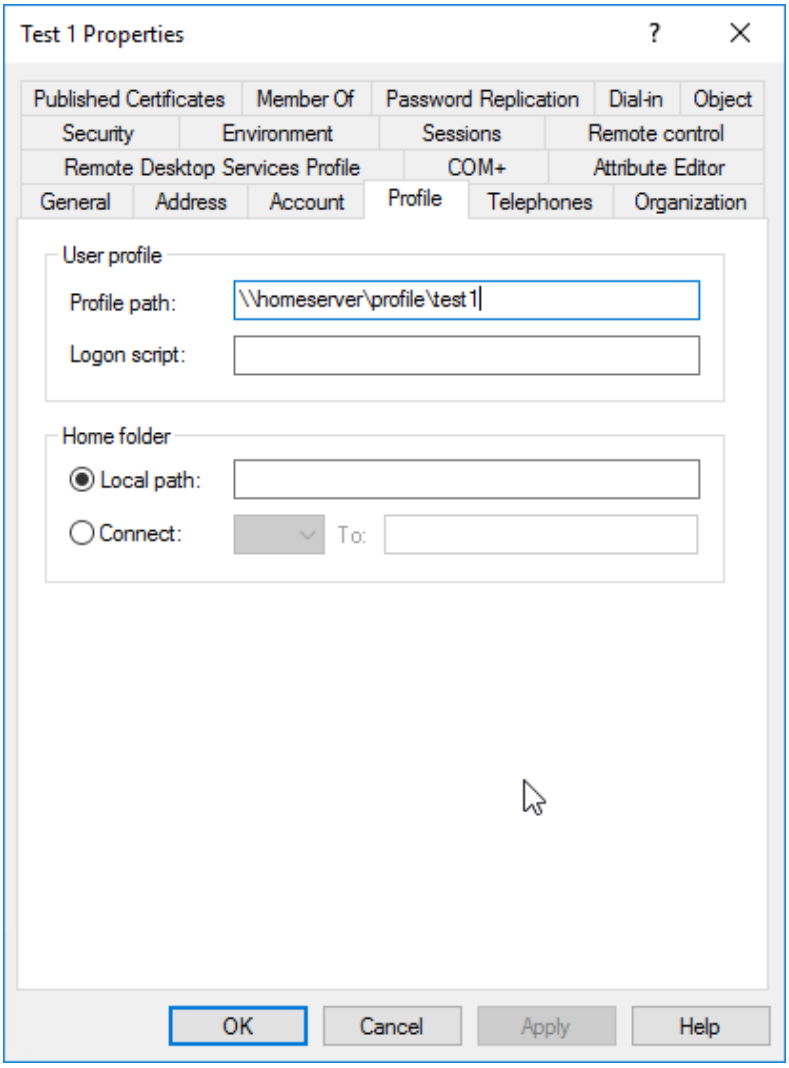
```
$shareName.SubString($shareName.Length-1,1)

                                                                    #
=================================================================
================================
   # If the last character is not a '\' then it is a regular
share. Then it is simple: call the
   # function 'Search-InTheFolder'
   #
   # If the last character is a '\' then only the servername
is given. So first find all the
    # shares on that server. After that, process all the
subshares.
                                                                    #
=================================================================
================================
   if ($LastCharacter -ne "\")
   {
    Search-InTheFolder -RootOfTheShare $shareName -numLevels
$NumberOfLevelsToSearch
   }
   Else
   {
     $arrShares = Get-NetShares -ComputerName $shareName
     ForEach($objShare in $arrShares)
     {
                                        $LastCharacter      =
($objShare.shi1_netname).SubString(($objShare.shi1_netname).Le
ngth-1,1)
      if($LastCharacter -ne "$")
      {
                                                                    #
=================================================================
================================
       # Ignore C$, D$, IPC$, NETADMIN$ etc.
       # This means that all the hidden shares are ignored.
                                                                    #
=================================================================
================================
       $ServerShareName = $shareName + $objShare.shi1_netname
            Get-FolderRights  -FolderNameToInvestigate
```

```
$ServerShareName
        Search-InTheFolder -RootOfTheShare $ServerShareName -
numLevels $NumberOfLevelsToSearch
      }
     }
   }
   $valCounter++
  }

#
================================================================
================================
# Output naar een CSV file
#
================================================================
================================

  $Global:Report | Sort-Object -Property FolderName,"AD Group"
| Export-Csv -path $CSVExportFile -NoTypeInformation -Encoding
ASCII
  Write-Host You can open the file $CSVExportFile now.
  Write-Host (Get-Date).ToString('T')  " Ended..."
```

Any feedback to improve this script is appreciated. You can download the scripts here:

1. Link to Inventory Permissions on Shares (v0.1)
2. Link to Inventory Permissions on Shares (v0.2)
3. Link to Inventory Permissions on Shares (v0.3)

Modify the users' profile path in Active Directory



Profile path in AD.

The customer I am working for asked me to write a script that removes the profile path from the users' profile in Active Directory. One of the requirements was to do it on a batch-by-batch basis. Thus not a big bang.

I decided to write a PowerShell script and use a .csv file as the input. The results should be written to a log file.

I created the following script:

```
<#
.NOTES
================================================================
================================
```

```
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Filename:         RemoveProfilePathFromUser (v02).ps1
================================================================
================================
.DESCRIPTION:

This script removes the profile path from the users' profile
in Active Directory.

.USAGE:

Create a CSV file with the following layout:

Userid
test1
test2

And mention the CSV file as a parameter -
FileWithUseridsInCSVFormat.

.VERSION HISTORY:
 v0.1:
   * Initial version.

 v.0.2:
   * Option -FileWithUseridsInCSVFormat has been added.
   * Added help text by the options.

 v.0.3:
   * Minor cosmetic changes.

#>

param
(
[Parameter(HelpMessage="CSV Filename that contains all the
userids that should be modified. If not mentioned than the
script name is used.")]
```

```
[String] $FileWithUseridsInCSVFormat=""
)

#
=================================================================
==================================
# Function block
#
=================================================================
==================================

Function Write-EntryToResultsFile
{

<#
.NOTES
=================================================================
==================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Write-EntryToResultsFile
=================================================================
==================================
.DESCRIPTION:

This function adds the success or failure information to the
array that contains the log
information.

#>
param
 (
  $strUserid,
  $ErrorMessage = ""
 )
 $Record            = [ordered] @{"Username" = "";"Error"= ""}
 $Record."Username" = $strUserid
 $Record."Error"    = $ErrorMessage
 $objRecord         = New-Object PSObject -Property $Record
```

```
 $Global:arrTable   += $objRecord
}

Function Process-User

<#
.NOTES
===========================================================
=================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Process-User
===========================================================
=================================
.DESCRIPTION:

Determine if the user can be modified.

#>
{
param
(
 $strUserid
)
    Try
       {
        $UN = Get-ADUser -Identity $strUserid
          Remove-ProfilePathFromUserProfileInAD -strUserid
$strUserid
       }
        Catch
       {
         Write-EntryToResultsFile -strUserid $strUserid -
ErrorMessage $_.Exception.Message
        Continue
       }

}
```

```
Function Remove-ProfilePathFromUserProfileInAD
{

<#
.NOTES
================================================================
================================
Created with:     Windows PowerShell ISE
Created on:       03-August-2018
Created by:       Willem-Jan Vroom
Organization:
Functionname:     Write-EntryToResultsFile
================================================================
================================
.DESCRIPTION:

This function adds the success or failure information to the
array that contains the log
information.

#>

  param
   (
    $strUserid
   )
   Try
      {
       Set-ADUser -Identity $strUserid -Clear profilePath
       Write-EntryToResultsFile -strUserid $strUserid
      }
       Catch
      {
         Write-EntryToResultsFile -strUserid $strUserid -
ErrorMessage $_.Exception.Message
       Continue
      }
}

#
================================================================
```

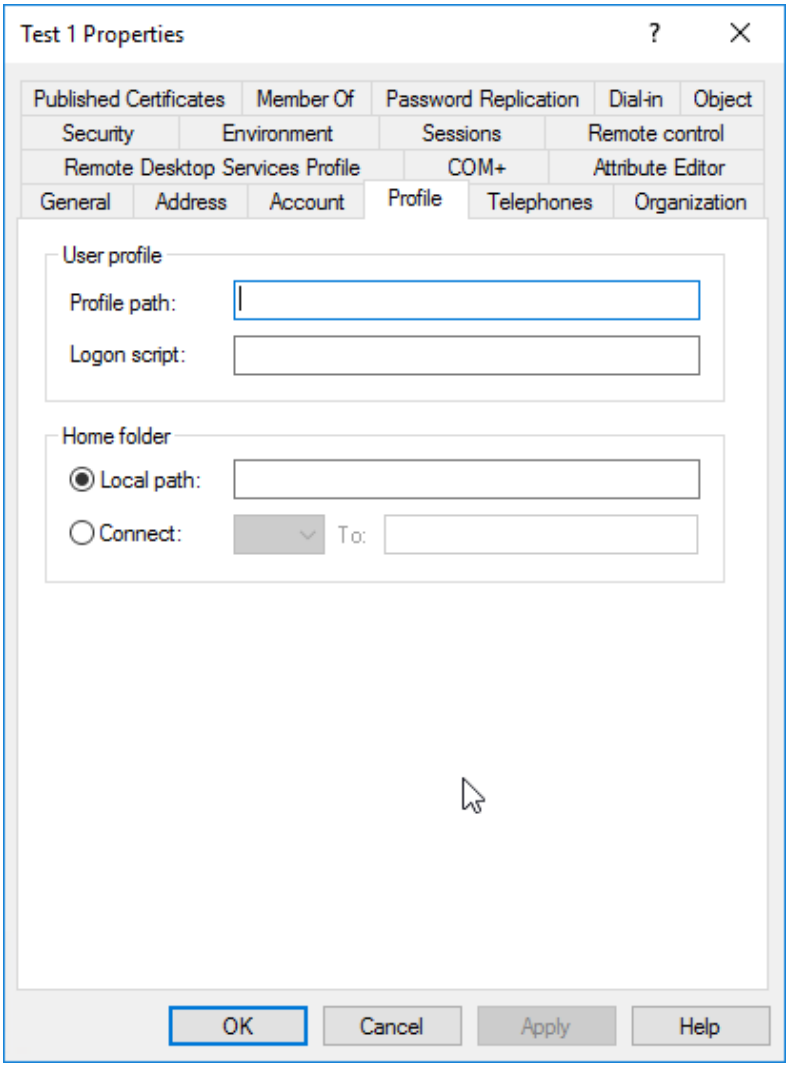
```
================================
# End function block
#
================================================================
================================

#
================================================================
================================
# Define the CSV Import File.
#
================================================================
================================

  $currentPath                        = Split-Path -parent
$MyInvocation.MyCommand.Definition
    $strCurrentFile                                       =
$MyInvocation.MyCommand.Name
  if($FileWithUseridsInCSVFormat.Length -eq 0)
   {
    $strCSVFileName                 = $strCurrentFile -Replace
".ps1",".csv"
   }
    else
   {
    if($FileWithUseridsInCSVFormat.ToLower().IndexOf(".csv") -
eq -1)
    {
     $FileWithUseridsInCSVFormat =+ ".csv"
    }
    $strCSVFileName = $FileWithUseridsInCSVFormat
   }

#
================================================================
================================
# Check if the string $strCSVFileName is a path. In that case,
nothing has to be done.
# In case it is not a path, then the current location should
be added.
#
```

```
================================================================
================================


  if (-not(Split-Path($strCSVFileName)))
  {
   $strCSVFileName = $currentPath + "\" + $strCSVFileName
  }

#
================================================================
================================
# Declares the variables.
#
================================================================
================================

  $valCounter                     = 1
  $Global:arrTable                = @()
  $Record                          = [ordered] @{"Username" =
"";"Error"= ""}

#
================================================================
================================
# Define the log file. This log file contains all the results.
#
================================================================
================================

    $strCSVLogFileSucces  =  $strCSVFileName  -Replace
".csv","_(Results).csv"
  If(Test-Path $strCSVLogFileSucces)
  {
  Remove-Item $strCSVLogFileSucces
  }

#
================================================================
================================
# Read the CSV file.
```

```
#
================================================================
================================

  if(Test-Path -LiteralPath $strCSVFileName)
  {
   $arrUserids = Import-Csv $strCSVFileName
  }
   Else
  {
    Write-Host "The import file $strCSVFileName does not
exists."
   Exit 1
  }

#
================================================================
================================
# Modify the users' profile path.
# Write the success or failure to the array with the results.
#
================================================================
================================

  Import-Module ActiveDirectory
  Clear-Host
  Write-Host (Get-Date).ToString('T')  "Started."
  ForEach ($objUserid in $arrUserids)
   {
     $strUserid = $objUserid.Userid
     if ($strUserid.Length -gt 0)
     {
        Write-Progress -Activity "Removing profile path from
user in Active Directory" -Status "Processing user $strUserid"
-PercentComplete ($valCounter / $arrUserids.Count * 100)
       Process-User -strUserid $strUserid
       Start-Sleep -s 1
       $valCounter ++
     }
       else
     {
```

```
        Write-Host "The input file $strCSVFileName has an
invalid layout. The column header should be named 'Userid'."
       Exit 1
     }
   }
  Start-Sleep -s 1

#
================================================================
================================
# Write the results to the csv file.
#
================================================================
================================

  If($Global:arrTable.Count -gt 0)
  {
     $Global:arrTable | Export-Csv $strCSVLogFileSucces -
NoTypeInformation
  }
   Else
  {
    Write-Host "Something went wrong while writing the logfile
$strCSVLogFileSucces. Maybe nothing to report..."
  }
Write-Host (Get-Date).ToString('T')  "Finished."
```

After running the script with the correct permissions, the profile path looks like:



Profile path after running the script.

The scripts can be downloaded in ZIP format:

1. Link to RemoveProfilePathFromUser (v0.1)
2. Link to RemoveProfilePathFromUser (v0.2)
3. Link to RemoveProfilePathFromUser (v0.3)