

# Windows mapped drives – what the hell is going on?

 [bloggingforlogging.com/2018/11/22/windows-mapped-drives-what-the-hell-is-going-on](https://bloggingforlogging.com/2018/11/22/windows-mapped-drives-what-the-hell-is-going-on)

November 22, 2018

WinObj - Sysinternals: [www.sysinternals.com](http://www.sysinternals.com)

File View Help

Name /	Type	SymLink
ACPI#FixedButton#28&daba3ff&1...	SymbolicLink	\Device\00000016
ACPI#GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000014
ACPI#GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000014
ACPI#GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000015
ACPI#GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000015
ACPI#PNP0303#48&e03a8448&0#{8...	SymbolicLink	\Device\00000019
ACPI#PNP0C0A#0#{72631e54-78a...	SymbolicLink	\Device\00000017
ACPI#PNP0C0A#0#{e849804e-c71...	SymbolicLink	\Device\00000017
ACPI#PNP0F03#48&e03a8448&0#{3...	SymbolicLink	\Device\0000001b
ACPI_ROOT_OBJECT	SymbolicLink	\Device\00000012
ahcache	SymbolicLink	\Device\ahcache
AUX	SymbolicLink	\DosDevices\COM1
C:	SymbolicLink	\Device\HarddiskVolume2
CON	SymbolicLink	\Device\ConDrv\Console
CONINS	SymbolicLink	\Device\ConDrv\CurrentIn
CONOUT\$	SymbolicLink	\Device\ConDrv\CurrentOut
Disk{9bbecca5-f9da-690b-c75d-76...	SymbolicLink	\Device\Harddisk0\DR0
DISPLAY#Default_Monitor#4&39f...	SymbolicLink	\Device\0000001e
DISPLAY#Default_Monitor#4&39f...	SymbolicLink	\Device\0000001e
DISPLAY1	SymbolicLink	\Device\Video0

Mapped drives have always been a curiosity for me, I've used them before in the past but usually come across an issue that forces me to abandon them. Alongside my curiosity, there has also been some demand in Ansible to be able to manage mapped drives and in my naivety I created a very basic module [win\\_mapped\\_drive](#) to do this. At the time it worked for what I needed to do and I had to use some hacks and workarounds to actually enumerate the existing mappings when running in WinRM. I should have read the warning signs then and given up but I pushed on ahead and the module was released with Ansible 2.4. We are now in the 2.8 development phase and there's been a few issues crop up on GitHub saying the module doesn't work as expected. I decided to take the time to look into the complex world of mapped drives and come up with a more satisfactory solution for managing them with Ansible.

The end result was [a complete rewrite of win\\_mapped\\_drive](#) and a really bad headache from trying to understand how this all fits together. I decided to try and put down what I learnt to help anybody who needs to deal with mapped drives as it isn't easy.

## What is a mapped drive

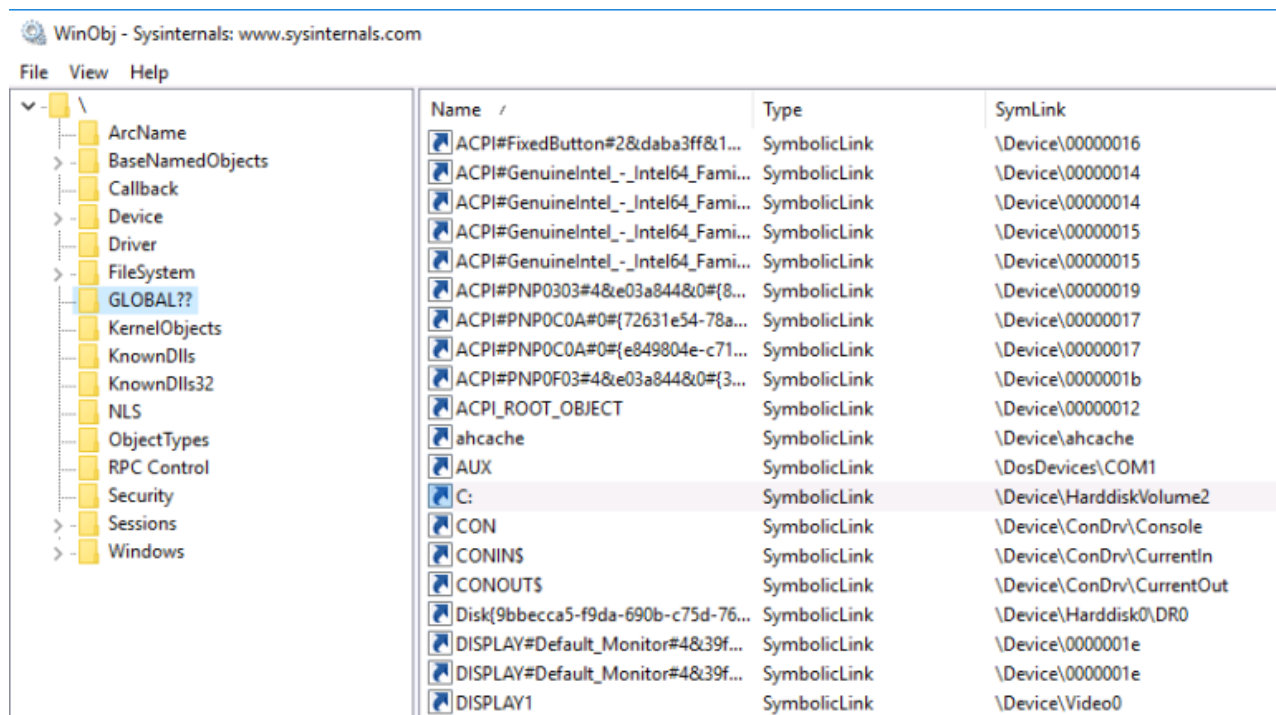
At a very high level, a mapped drive is a redirected network resource to a local device, e.g. redirects a UNC path `\\SERVER\share` to a local drive `Z:`. This allows a user to access network resources like they would with a local file. In modern days this is less of an issue as you can use the full UNC path to access a file but there are still other benefits

for using mapped drives from an end user perspective. Delving a bit deeper, a mapped drive is actually a DOS Device Name that points to an NT device that does all the redirection for you.

## DOS Device Names

You might be saying “DOS? I thought DOS was a bygone product that was removed from existence with the introduction of Windows NT”. This is mostly true but its legacy still lives on and DOS Devices is a concept that was transferred to Windows NT. A DOS device is basically an object name that exists in the NT Object Manager namespace and is used by various file management functions to redirect DOS names to the underlying NT object. In a more simplistic manner, a DOS device is an object that represents a traditional DOS device name, such as **NUL**, **CON**, **C:**, and links those object names to the actual NT object that handles the IO functions. For example, the path **C:\Windows\System32\cmd.exe** is expanded to **\DosDevices\C:\Windows\System32\cmd.exe**, **\DosDevices\C:** is a symlink to **\Device\HarddiskVolumex** which means **cmd.exe** can be accessed at the NT path **\Device\HarddiskVolumex\Windows\System32\cmd.exe**.

To add some more complexity to these devices, there exists a local and global scope to a device where a global scope is accessible to all users on the host and a local scope is only accessible to the logon session that created it. A physical hard drive device, like **C:**, is a global device and is accessible to all users on the host. We can see this by using the program WinObj, under **\GLOBAL??** we can see many objects including our **C:** device and where it links to:



Name	Type	SymLink
ACPI\FixedButton#28&daba3ff&1...	SymbolicLink	\Device\00000016
ACPI\GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000014
ACPI\GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000014
ACPI\GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000015
ACPI\GenuineIntel_-_Intel64_Fami...	SymbolicLink	\Device\00000015
ACPI\PNP0303#48&e03a844&0#{8...	SymbolicLink	\Device\00000019
ACPI\PNP0C0A#0#{72631e54-78a...	SymbolicLink	\Device\00000017
ACPI\PNP0C0A#0#{e849804e-c71...	SymbolicLink	\Device\00000017
ACPI\PNP0F03#48&e03a844&0#{3...	SymbolicLink	\Device\0000001b
ACPI_ROOT_OBJECT	SymbolicLink	\Device\00000012
ahcache	SymbolicLink	\Device\ahcache
AUX	SymbolicLink	\DosDevices\COM1
C:	SymbolicLink	\Device\HarddiskVolume2
CON	SymbolicLink	\Device\ConDrv\Console
CONIN\$	SymbolicLink	\Device\ConDrv\CurrentIn
CONOUT\$	SymbolicLink	\Device\ConDrv\CurrentOut
Disk\9bbecca5-f9da-690b-c75d-76...	SymbolicLink	\Device\Harddisk0\DR0
DISPLAY#Default_Monitor#4&39f...	SymbolicLink	\Device\0000001e
DISPLAY#Default_Monitor#4&39f...	SymbolicLink	\Device\0000001e
DISPLAY1	SymbolicLink	\Device\Video0

We can also see some other global DOS devices like **CON**, **AUX** and the relevant NT object they link to. Local DOS devices are scoped to a specific logon session ID and a device name in the local scope takes priority over a global DOS device. More details on the local vs global scopes can be found here. When a user creates a new mapped drive the DOS

device that is created will exist under the local scope and thus only accessible to that particular logon session. The only exception to this rule is when running or impersonating the **SYSTEM** account, a mapped drive will be created in the global scope and thus accessible to all the users on the host.

Having DOS devices scoped to an individual logon session is pretty much the main source of confusion of when mapped drives are accessible.

## Logon sessions

---

Not to be confused with a Windows Session, a logon session is created whenever the Local Security Authority (LSA) processes a new logon request. A logon can be anything, such as;

- a user logging in directory on the host
- through an RDP logon
- a scheduled task with explicit user credential
- using runas.exe
- SMB network logon
- a WinRM process
- using some Win32 APIs like CreateProcessWithLogon, LogonUser, LsaLogonUser

When LSA processes the logon, it will build the access token based on the rights and groups that are assigned to the account and create a unique identifier that links the access token to a unique, to LSA, logon session identifier. This ID is stored in the access token and can be queried by anyone who has the rights to but cannot be changed (well not officially). Complicating matter further, an access token has a one to one relation to a logon session but a user may have one or two access tokens that are linked together to a single logon event.

## Token Elevation Types

---

Before Windows Vista and the introduction of User Account Control (UAC), a user had only one token which contains all the groups and privileges that it had the rights to. With Windows Vista, an access token now has a flag under the TOKEN\_ELEVATION\_TYPE enum with one of the following values;

- **TokenElevationTypeDefault**: This is like the Windows XP days, the token is not linked to anything else and contains all the groups and privileges that is assigned to the user
- **TokenElevationTypeFull**: The token is part of a linked pair and contains all the groups and privileges for the user
- **TokenElevationTypeLimited**: The token is part of a linked pair and contains a limited set of groups and privileges for the user

The **TokenElevationTypeDefault** is used in, but not limited to, these scenarios;

- UAC is disabled
- You are using the `BUILTIN\Administrators` account and `FilterAdministratorToken` is set to `Disabled` (default behaviour)
- You are logging in with a standard user account
- You are logging in through a network logon, like SMB or WinRM, with a domain account
- You are logging in through a network logon with a local account and `LocalAccountTokenFilterPolicy` is set to `1`

If you are using an admin account and one of the above does not match your setup then LSA will produce two access tokens and logon sessions that are linked together for a single logon. The default access token will have a type of `TokenElevationTypeLimited` while the linked token will have a type of `TokenElevationTypeFull`. To find out what the current process token type is, run the following PowerShell script;

```

Add-Type -TypeDefinition @"
using Microsoft.Win32.SafeHandles;
using System;
using System.ComponentModel;
using System.Runtime.ConstrainedExecution;
using System.Runtime.InteropServices;
using System.Security.Principal;

namespace PInvoke
{
    internal class NativeMethods
    {
        [DllImport("kernel32.dll", SetLastError = true)]
        public static extern bool CloseHandle(
            IntPtr hObject);

        [DllImport("kernel32.dll")]
        public static extern SafeNativeHandle GetCurrentProcess();

        [DllImport("advapi32.dll", SetLastError = true)]
        public static extern bool GetTokenInformation(
            SafeNativeHandle TokenHandle,
            UInt32 TokenInformationClass,
            SafeMemoryBuffer TokenInformation,
            UInt32 TokenInformationLength,
            out UInt32 ReturnLength);

        [DllImport("advapi32.dll", SetLastError = true)]
        public static extern bool OpenProcessToken(
            SafeNativeHandle ProcessHandle,
            TokenAccessLevels DesiredAccess,
            out SafeNativeHandle TokenHandle);
    }

    internal class SafeMemoryBuffer : SafeHandleZeroOrMinusOneIsInvalid
    {
        public SafeMemoryBuffer() : base(true) { }
        public SafeMemoryBuffer(int cb) : base(true)
        {
            base.SetHandle(Marshal.AllocHGlobal(cb));
        }
        public SafeMemoryBuffer(IntPtr handle) : base(true)
        {
            base.SetHandle(handle);
        }

        [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
        protected override bool ReleaseHandle()
        {
            Marshal.FreeHGlobal(handle);
            return true;
        }
    }

    internal class SafeNativeHandle : SafeHandleZeroOrMinusOneIsInvalid
    {

```

```

    public SafeNativeHandle() : base(true) { }
    public SafeNativeHandle(IntPtr handle) : base(true) { this.handle =
handle; }

    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
    protected override bool ReleaseHandle()
    {
        return NativeMethods.CloseHandle(handle);
    }
}

public enum TokenElevationType
{
    TokenElevationTypeDefault = 1,
    TokenElevationTypeFull,
    TokenElevationTypeLimited
}

public class AccessToken
{
    public static TokenElevationType GetTokenElevationType()
    {
        using(SafeNativeHandle hProcess = NativeMethods.GetCurrentProcess())
        {
            SafeNativeHandle hToken;
            NativeMethods.OpenProcessToken(hProcess, TokenAccessLevels.Query,
out hToken);

            if (hToken.IsInvalid)
                throw new Win32Exception();

            using (hToken)
            {
                UInt32 tokenLength;
                NativeMethods.GetTokenInformation(hToken, 18, new
SafeMemoryBuffer(IntPtr.Zero), 0, out tokenLength);

                using (SafeMemoryBuffer tokenInfo = new
SafeMemoryBuffer((int)tokenLength))
                {
                    if (!NativeMethods.GetTokenInformation(hToken, 18,
tokenInfo, tokenLength, out tokenLength))
                        throw new Win32Exception();
                    return
(TokenElevationType)Marshal.ReadInt32(tokenInfo.DangerousGetHandle());
                }
            }
        }
    }
}

'@
[PInvoke.AccessToken]::GetTokenElevationType()

```

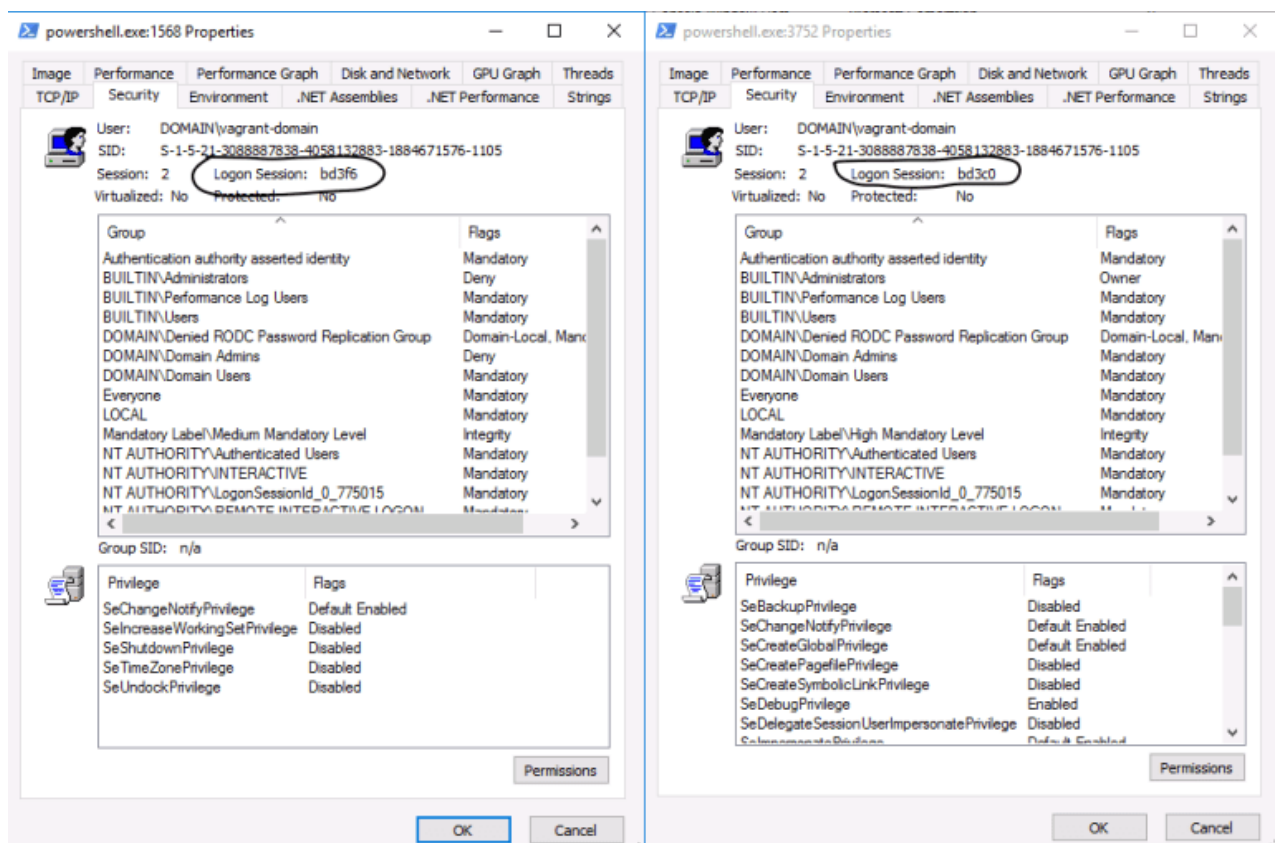
So why is this important? Having a split token means there are two DOS device local scopes a user may use and each are separate from each other. In laymans terms an admin process will not be able to see drives mapped under a limited token and vice versa. This gets even more complicated when persistence is added into the mix but I'll cover that later. If you received `TokenElevationTypeDefault` then you don't have to worry about the next few sections.

## EnabledLinkedConnections

While local DOS devices are scoped to the current logon session, Windows does expose a policy to help deal with mapped drives and split tokens. The `EnableLinkedConnections` can be set and it will change the behaviour when running with the `TokenElevationTypeFull` and `TokenElevationTypeLimited` elevation types to be more like `TokenElevationTypeDefault`. When it is set, any drives created in one logon session will then be added to the logon session of the split token. This makes things a bit more uniform across the various token elevation types but does have some potential security implications. The rest of this post will continue under the assumption that this policy is either not defined or is disabled.

## Logon session with a split token

Because it is the more common scenario in a locked down host I want to delve a bit deeper into split tokens and how it affects mapped drives. I mentioned above that with a split token, LSA has created two different access tokens each with their own unique logon session ID as we can see from the output below;



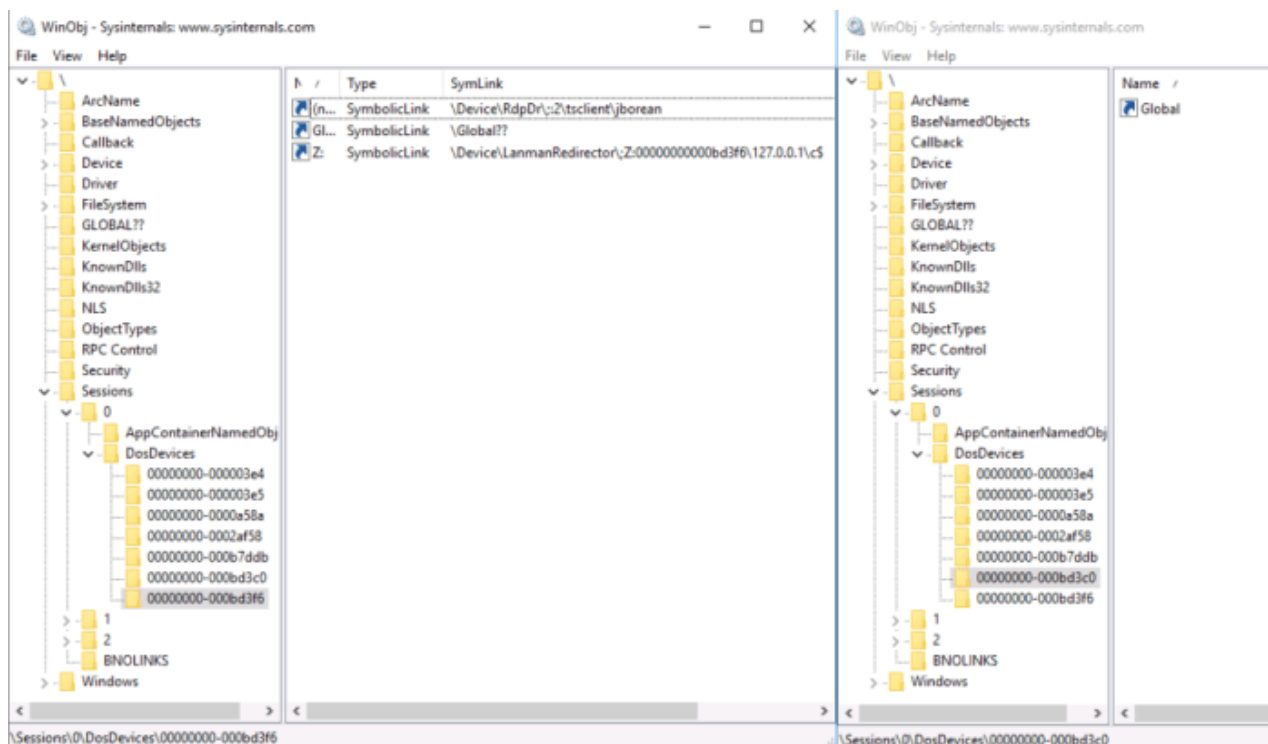


We can also see different groups and privileges in each token

In this example, I've started a PowerShell process normally and another by right clicking and selecting **Run as administrator**. We can see the limited access token has a logon session of **bd3f6** while the "linked" admin token has a logon session of **bd3c0**. As I stated above, a locally scoped DOS device is only accessible for the logon session it was created for, because of this my process running with an admin token is unable to see DOS devices created by the limited token and vice versa. To prove my point, run **net use Z: \\127.0.0.1\c\$** in the limited PowerShell session then run **net use** in both processes.

Limited (left) vs Admin (right)

We can see that the limited PowerShell process is able to see our newly mapped drive while the admin PowerShell process cannot due to the isolation of DOS devices per logon session. To further prove our point, we can use WinObj to view the local DOS Devices. We know global devices are located in **\GLOBAL??** but to find local devices we need to look at **\Sessions\0\DosDevices\{logon session id}**. In our case our limited device location is **\Sessions\0\DosDevices\00000000-000bd3f6** while our admin device location is **\Sessions\0\DosDevices\00000000-000bd3c0**.



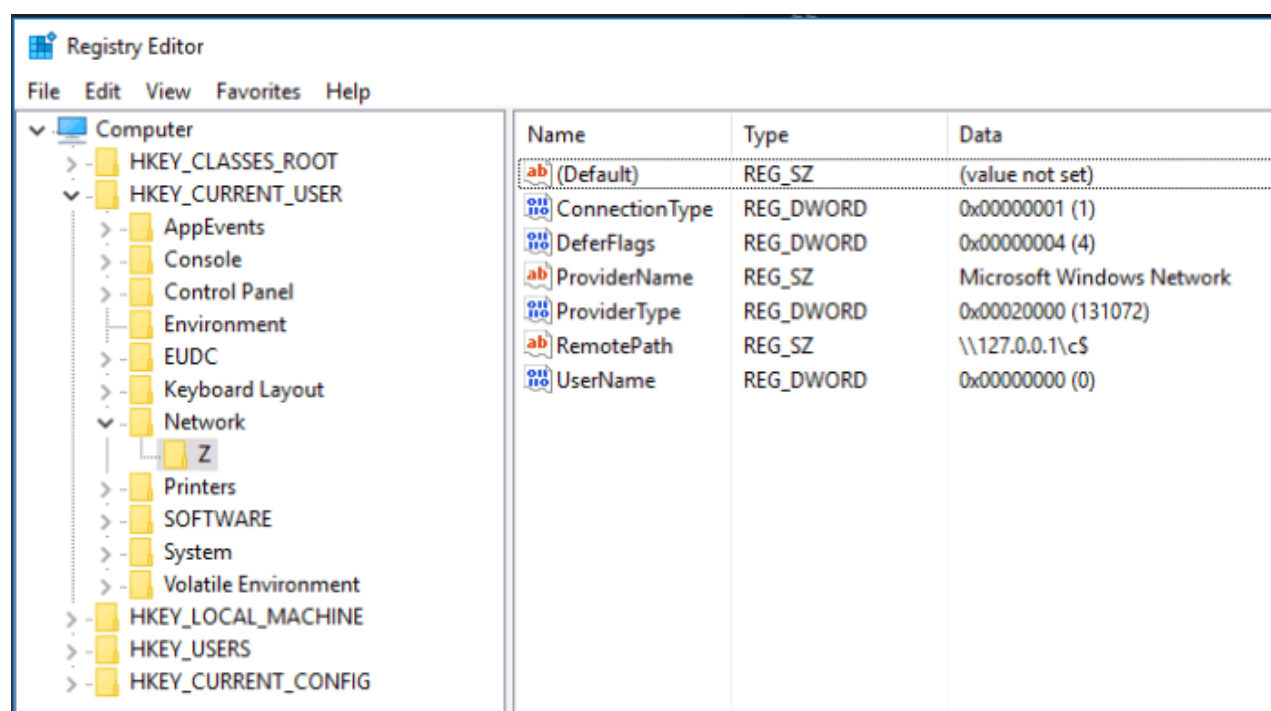


So we can see that created mapped drives exist only in the logon session it was creating in. Having the **EnabledLinkedConnections** policy set changes this behaviour to have it defined in both logon sessions.

This explains how mapped drives are created in an adhoc fashion but another areas we haven't talked about is persisting these mapped drives across new logons and reboots.

## Drive persistence

Using the **net use** command above we have created a mapped drive for the logon session when the user logs off and back on, the logon session ID will be different and it will no longer be able to access the mapped drive from before. Windows offers a way to persist these drives so that the user can continue to use them even after a new logon or reboot. To try this out, run **net use \* /delete** to clear out any existing mappings and then run **net use Z: \\127.0.0.1\c\$ /persistent:yes** in our limited PowerShell process and then **net use Y: \\127.0.0.1\c\$ /persistent:yes** in our admin PowerShell process. When using the **/persistent:yes** option, **net use** will create a registry key at **HKCU:\Network\{letter}** and store the info required to rebuild the mapped drive when needed.



The values match up to the WNetAddConnection2 API

We can see our drive mapping for **Z:** but not for **Y:** which is an ominous sign and just like before our limited PowerShell process can only see the **Z:** drive whereas our admin PowerShell process can only see our **Y:** drive. Log off and back on again and we will continue to see that **Z:** is mapped based on the values in the registry key and is still accessible from other limited processes like Windows Explorer. When using **net use** in an admin process we can no longer see **Y:** so that was not persisted.

In short, only drives mapped with an access token of the elevation type `TokenElevationTypeDefault` or `TokenElevationTypeLimited` can create a persisted mapped drive whereas `TokenElevationTypeFull` cannot persist a mapped drive. Having the `EnableLinkedConnections` set means `TokenElevationTypeFull` access tokens will be able to see and persisted mapped network drives like the other token elevation types but this isn't set by default.

So we know why admin processes are unable to see and persist mapped drives by default we still need to figure out why we can't do the same for other limited interactive processes, like ones spawned from `runas.exe`. These processes will run with the limited access token but when using `Get-PSDrive` we can no longer see the actual mapped drive while `net use` shows it as Unavailable.

```

PS C:\Users\vagrant-domain> net use
New connections will be remembered.

Status      Local      Remote      Network
-----
OK          Z:         \\127.0.0.1\c$  Microsoft Windows Network
The command completed successfully.

PS C:\Users\vagrant-domain> Get-PSDrive

Name      Used (GB)  Free (GB)  Provider  Root
-----
Alias
C          28.22     10.50     FileSystem C:\
Cert
Env
Function
HKCU
HKLM
Variable
WSMan
Z          28.22     10.50     FileSystem \\127.0.0.1\c$

PS C:\Users\vagrant-domain> runas.exe /user:vagrant-domain@DOMAIN.LOCAL powershell
Enter the password for vagrant-domain@DOMAIN.LOCAL:
Attempting to start powershell.exe as user "vagrant-domain@DOMAIN.LOCAL" ...
PS C:\Users\vagrant-domain> $pid
1860
PS C:\Users\vagrant-domain>

```

powershell.exe:1860 Properties

User: DOMAIN\vagrant-domain  
SID: S-1-5-21-308887838-4058132883-1884671576-1105  
Session: 3 Logon Session: 3a40ac  
Virtualized: No Protected: No

Group	Flags
Authentication authority asserted identity	Mandate
BUILTIN\Administrators	Deny
BUILTIN\Performance Log Users	Mandate
BUILTIN\Users	Mandate
DOMAIN\Denied RODC Password Replication Group	Domain-Deny
DOMAIN\Domain Admins	Deny
DOMAIN\Domain Users	Mandate
Everyone	Mandate

Group SID: n/a

Privilege	Flags
SeChangeNotifyPrivilege	Default Enabled
SeIncreaseWorkingSetPrivilege	Disabled
SeShutdownPrivilege	Disabled
SeTimeZonePrivilege	Disabled
SeUndockPrivilege	Disabled

```

PS C:\Windows\system32> net use
New connections will be remembered.

Status      Local      Remote      Network
-----
Unavailable Z:         \\127.0.0.1\c$  Microsoft Windows Net
Unavailable \\TSCLIENT\jborean Microsoft Terminal Se
The command completed successfully.

PS C:\Windows\system32> Get-PSDrive

Name      Used (GB)  Free (GB)  Provider  Root
-----
Alias
C          28.22     10.50     FileSystem C:\
Cert
Env
Function
HKCU
HKLM
Variable
WSMan
Z          28.22     10.50     FileSystem \\127.0.0.1\c$

PS C:\Windows\system32> $pid
4632
PS C:\Windows\system32>

```

powershell.exe:4632 Properties

User: DOMAIN\vagrant-domain  
SID: S-1-5-21-308887838-4058132883-1884671576-1105  
Session: 3 Logon Session: 6f5d46  
Virtualized: No Protected: No

Group	Flags
Authentication authority asserted identity	Mandate
BUILTIN\Administrators	Deny
BUILTIN\Performance Log Users	Mandate
BUILTIN\Users	Mandate
DOMAIN\Denied RODC Password Replication Group	Domain-Deny
DOMAIN\Domain Admins	Deny
DOMAIN\Domain Users	Mandate
Everyone	Mandate

Group SID: S-1-2-0

Privilege	Flags
SeChangeNotifyPrivilege	Default Enabled
SeIncreaseWorkingSetPrivilege	Disabled
SeShutdownPrivilege	Disabled
SeTimeZonePrivilege	Disabled
SeUndockPrivilege	Disabled

Same Windows session but different logon session

When looking at the access token for the `runas.exe` spawned process we can see that while it has the same Windows Session, groups, and privileges, the logon session ID is different. Because it is part of a different logon session it is no longer able to see the locally scoped mapped drives of my current interactive token. The only reason why they come up as unavailable with `net use` is because that tool is scanning the registry for persistent configurations but then cannot find the locally scoped DOS device with that name. You can even prove that no devices have been created using `WinObj` by checking under `\Sessions\0\DosDevices\{logon session id}`. When running a task through

Ansible, or any other WinRM process like `Enter-PSSession`, you will see the exact same behaviour which indicates the automatic mapping on a logon is not done when LSA creates a new logon session but by something else.

I don't have access to the Windows source code to confirm this but after reading the [Windows Internals Book Part 1](#) book this is what I believe happens;

- Interactive logons are handled by the `Winlogon` executable of the Windows Session
- This may be an existing session when logging on through the console or a new session when logging on through RDP
- Part of the Winlogon process is to notify the [Multiple Provider Router](#) (MPR) that a new logon has occurred
- MPR will then check the registry and load the various network providers, one of them being the `Microsoft Windows Network` provider
- The `Microsoft Windows Network` provider will scan the registry at `HKCU:\Network` and then attempt to map each mapped drive configuration for the current logon token by using `WNetAddConnection2W`
- If the logon token is `TokenElevationTypeLimited` and the `EnableLinkedConnections` policy is set, then the network provider will also map the same drive to the split token with the `TokenElevationTypeFull` type

The key component that starts this whole process is `Winlogon` and because `runas.exe`, WinRM, scheduled tasks don't use `Winlogon`, the persistent drive mapping is never automatically run. Because of this it will not be possible for applications that use WinRM, like Ansible, to automatically access the user's mapped drives. They can manually call `net use` or read the registry and call `WNetAddConnection2W` but this will not be handled by Windows automatically.

## Credentials

---

I've covered mapped drives and how they are defined and used within Windows but one key component of mapped drives is the authentication process. Here is the order in which credentials are used when connecting to a network resource;

- When calling `WNetAddConnection2W`, it will use the credentials specified by `lpUserName` and `lpPassword` if set
- If the current access token has access to the user's Credential store and a credential exists for the remote target then that is used
- If the current access token has access to the user's credentials then it will use those. This is the typical interactive or RDP logon scenario
- No credentials could be accessed so Windows will authenticate as an anonymous user

I've covered logon types and credentials within WinRM in my [Demystifying WinRM](#) blog if you want to learn more about logon types, but typically a WinRM connection will not have access to the user's credentials so the last scenario applies. Ansible does offer a few

different ways to bypass the credential limitation which will allow a WinRM process access delegate its credential or access to the credential vault. These include;

- Using CredSSP as an authentication option, this will allow the user to access its credential vault as well as delegate its own credentials
- Using Kerberos with delegation to allow the user to delegate its own credentials
- Using Ansible become to create a pseudo interactive session with access to its credential vault and delegation rights

When using the `net use Z: \\server\share /user:username password /persistence:yes` command, Windows actually calls `WNetAddConnection2W` API with the `lpUserName` and `lpPassword` set based on our input so it's able to authenticate with the network resource. If successful the registry key at `HKCU:\Network\Z` will be populated with the details to remap on the next logon and the `UserName` property will be set to the value of `lpUserName`. We can continue to use the mapped drive with these credentials for the current logon session but as soon as we log off and back on it will fail to connect. Through trial and error I have found that while `WNetAddConnection2W` will use the credentials to create the initial logon session's mapped drive, the `lpPassword` value is not stored anywhere so subsequent logons will fail as Windows is sending a blank password as per this security event log entry.

**Security** 2 Events

Keywords	Date and Time	Source	Event ID	Task Category
Audit Failure	11/22/2018 4:06:53 AM	Microsoft Windows security auditing.	4625	Logon
Audit Succ...	11/22/2018 4:06:43 AM	Eventlog	1102	Log clear

Event 4625, Microsoft Windows security auditing.

**General** **Details**

An account failed to log on.

Subject:

- Security ID: NULL SID
- Account Name: -
- Account Domain: -
- Logon ID: 0x0

Logon Type: 3

Account For Which Logon Failed:

- Security ID: NULL SID
- Account Name: vagrant-domain@DOMAIN.LOCAL
- Account Domain: -

Failure Information:

- Failure Reason: Unknown user name or bad password.
- Status: 0xc000006d
- Sub Status: 0xc000006a

Process Information:

- Caller Process ID: 0x0
- Caller Process Name: -

Network Information:

- Workstation Name: SERVER2016
- Source Network Address: 192.168.56.16
- Source Port: 49879

Detailed Authentication Information:

- Logon Process: NtLmSsp
- Authentication Package: NTLM
- Transited Services: -
- Package Name (NTLM only): -
- Key Length: 0

0xc000006a == STATUS\_WRONG\_PASSWORD

Because of this issue, the only way I've found to automatically map a network drive that requires credentials is to store the required credential in the Windows Credential Manager for each user who needs that mapped drive. This can be done through the GUI but the `cmdkey` executable can be used to achieve the same thing. In this case I would run `cmdkey.exe /add:<server> /user:<username> /pass:<password>` before running my `net use`.



```
Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\vagrant>cmdkey.exe /add:server2008-x64 /user:vagrant-domain@DOMAIN.LOCAL /pass
Enter the password for 'vagrant-domain@DOMAIN.LOCAL' to connect to 'server2008-x64':

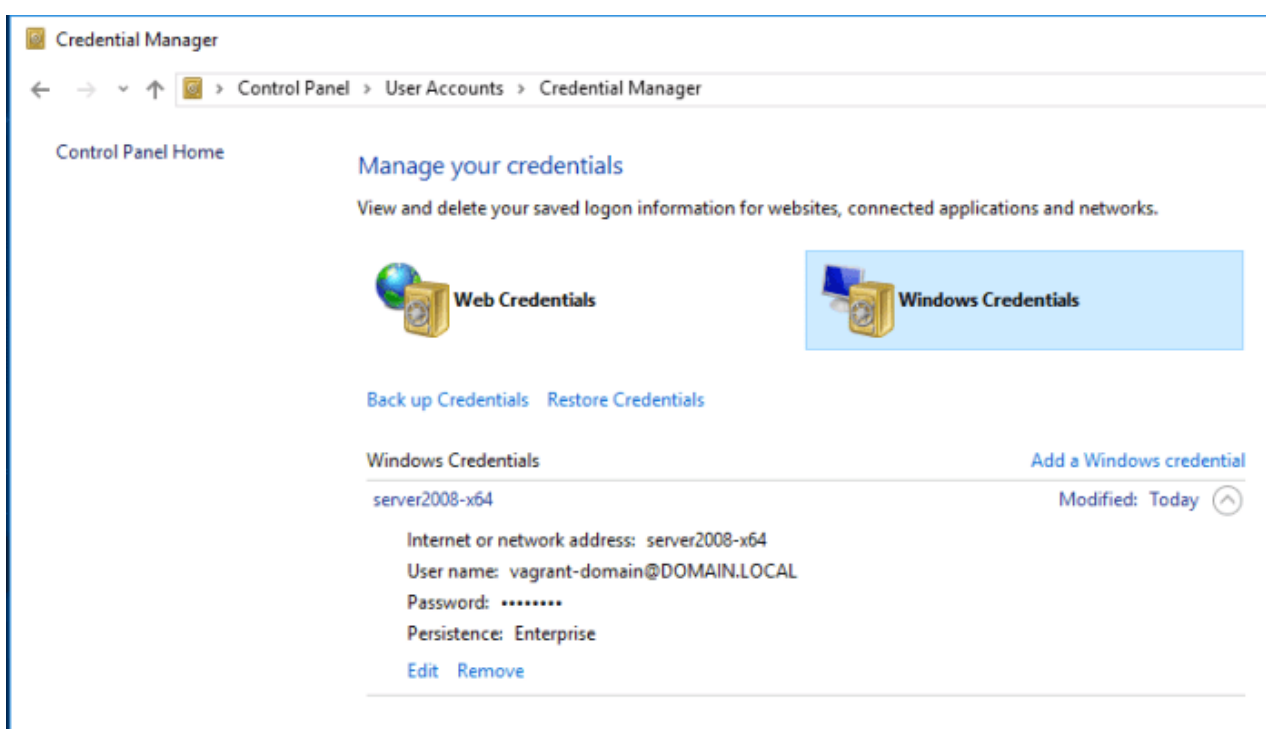
CMDKEY: Credential added successfully.

C:\Users\vagrant>net use Z: \\server2008-x64\temp /persistent:yes
The command completed successfully.

C:\Users\vagrant>
```

To prompt for a pass, omit the password after /pass

I can now log off and back on and have my mapped drive automatically connect with the credentials specified as well as manually manage the credentials in the Credential Manager.



Credentials are stored per user account so you cannot share credentials or create a global credential and each network resource can only have one credential assigned to it. The credentials themselves are stored in a reversible encrypted format that can only be read by a LSA authentication providers, in reality it's really trivial to use an application like mimikatz to dump passwords from the Credential Manager. The best defense is to not save credentials at all and just input them when it's needed but if you need to save credentials then make sure the account isn't used for anything else.

## Global mapped drives

I briefly touched on the global scope for network drives but I've mostly left this alone as the majority of the work focuses on per user mapped drives. It is possible to create a mapped drive that works for accounts on the current host and this is done by creating a globally scoped DOS device. Creating a globally scoped DOS device is as simple as running `net use` as the system account but having that persist on a reboot involves

modifying the registry at **HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\DOS Devices**. By creating a new string property where the name is the drive letter and the value is a NT Object path you want to redirect it to, Windows will automatically create this global DOS device during the **smss.exe** startup process. You can use this handy PowerShell cmdlet to build the Windows Object path and set the registry key for you.

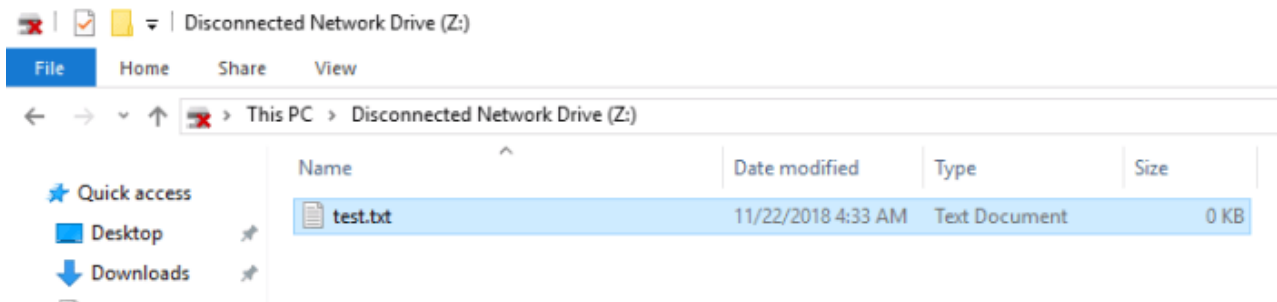
```
Function New-GlobalMappedDrive {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$true)][String]$Drive,
        [Parameter(Mandatory=$true)][String]$Path,
        [Switch]$Force
    )
    if ($Drive -notmatch "^[a-zA-z]{1}[:]?$") {
        throw [System.ArgumentException]"Drive must either be a single letter or a
letter with a :"
    } elseif ($Drive.Length -eq 1) {
        $Drive = "$($Drive):"
    }

    if ((-not [System.IO.Path]::IsPathRooted($Path)) -or (-not
$Path.StartsWith("\\"))) {
        throw [System.ArgumentException]"Path must be a UNC path starting with \"
    }

    $reg_key = "HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\DOS
Devices"
    # The SYSTEM account always has a logon session of 00000000-0000003e7
    $dos_path =
"\Device\NanmanRedirector\;$($Drive)000000000000003e7\$($Path.Substring(2))"
    $existing_property = Get-ItemProperty -Path $reg_key -Name $Drive -ErrorAction
SilentlyContinue

    if ($null -ne $existing_property) {
        $existing_dos_path = $existing_property.$Drive
        if ($existing_dos_path -eq $dos_path) {
            Write-Verbose -Message "The Global DOS Device '$Drive' is already
pointing to '$Path'"
        } elseif (-not $Force) {
            Write-Error -Message "Global DOS Device '$Drive' already exists, use -
Force to override"
        } else {
            Write-Verbose -Message "The Global DOS Device '$Drive' will be
repointed from '$existing_dos_path' to '$dos_path'"
            Set-ItemProperty -Path $reg_key -Name $Drive -Value $dos_path > $null
        }
    } else {
        Write-Verbose -Message "Creating new Global DOS Device '$Drive' pointing
to '$Path'"
        New-ItemProperty -Path $reg_key -Name $Drive -Value $dos_path > $null
    }
}
```

On the next reboot you will see the mapped drive in Windows Explorer that is disconnected but you can still double click and open it. I'm not sure why it shows up as disconnected by I believe it may be because the state of a network drive is handled by the actual network provider whereas we have bypassed it entirely by defining the global DOS device manually. Even better, since we've created a global DOS device, we should now be able to see this drive in a WinRM task as we are no longer restricted to an individual logon session.



We can still access the drive even though it shows up as disconnected

If the network resource requires authentication with a different account, the same rules that apply to a locally scoped DOS device also apply to a globally scoped one. That is, the user account must have a credential for that network host defined in it's own credential vault as there is no global credential vault that I'm aware of.

## Ansible

Throughout this post I've mostly covered Windows only topics but the main reason why I worked through this was to be able to easily manage network drives in Ansible. If you were able to use the devel branch in Ansible, or 2.8 once released, you can simply create a network drive with the following task

```
- name: create a drive with no authentication
  win_mapped_drive:
    letter: Z
    path: \\server\share
    state: present
```

This achieves the same result as running `net use Z: \\server\share /persistent:yes` but it includes all the idempotency checks and abstracts all the complex code required to achieve this through WinRM. If you need to create a mapped drive that requires authentication you can achieve this in two tasks;

```

- name: create a credential for the network resource
  win_credential:
    name: server
    type: domain_password
    username: username
    secret: password
    state: present
  # become is required to access the credential manager
  become: yes
  become_method: runas
  vars:
    # this is not the credential username/pass but the user
    # who's vault you want to save the credential in
    ansible_become_user: '{{ ansible_user }}'
    ansible_become_pass: '{{ ansible_password }}'

- name: create a mapped drive that requires authentication
  win_mapped_drive:
    letter: Z
    path: \\server\share
    state: present
  # unless running with CredSSP, you need to run this task
  # with become so it can access the credentials above
  become: yes
  become_method: runas
  vars:
    ansible_become_user: '{{ ansible_user }}'
    ansible_become_pass: '{{ ansible_password }}'

```

The first task will use create a Windows Credential for the network host `server` under the become user's credential vault and the second task will use those credentials when saving the mapped drive. Any future interactive logon sessions for that user will be able to see that drive and it will automatically use the credentials that were created in the first task.

The magic behind all this is using Ansible's `become` process to change the logon type from `network` to `interactive` so that it can access the user's credentials. Historically using become with `win_mapped_drive` would only work if the account did not have a split access token but the recent refactor is able to handle this scenario by impersonating the limited token when managing the network resources.

Say you wish to create a mapped drive without saving the credentials you can run the following

```

- name: create a mapped drive with temporary credentials
  win_mapped_drive:
    letter: Z
    path: \\server\share
    state: present
    username: username
    password: password

```

This will use `username` and `password` for the initial connection test but unlike `net use`, it will not save the `username` in the registry. We can see that `become` is not required in this case as we don't need access to the Credential Manager for authentication. The mapped drive will be available to any future interactive logons it will just require manual authentication by the user.

The last scenario would be creating a globally scoped mapped drive. If you wish to create a global mapped drive that will exist until the next reboot you can run this task

```
- name: create a global mapped drive without persistence
  win_mapped_drive:
    letter: Z
    path: \\server\share
    state: present
    # While this isn't always required, it may need explicit
    # credentials for the initial authentication
    username: username
    password: password
    become: yes
    become_method: runas
    become_user: SYSTEM
```

If you wish to create a global mapped drive that persists on a reboot you can edit the registry key like so

```
- name: create a global mapped drive with persistence
  win_regedit:
    path: HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\DOS Devices
    name: 'Z:'
    data: '\Device\LanmanRedirector\;Z:000000000000003e7\server\share'
    type: string
    state: present
```

The first example runs the code under the `SYSTEM` account which is the same as running `net use` under the `SYSTEM` account but as we've gone across it before this will not persist on a reboot. You don't have to map the drive using the `SYSTEM` account before creating the registry entry, the first task is only necessary if you want it available immediately.

## TLDR;

---

I've talked about some pretty low level Windows concepts in this post and thought it best to try and sum it all up for anyone just wanting a quick overview.

- mapped drives are created per logon session, the only exception is a mapped drive created by the `SYSTEM` account which is global
- a global mapped drive can be created by running the mapping process as the `SYSTEM` account
- a global mapped drive will only exist until the host is rebooted
- you can define global mapped drives that are mapped when Windows starts up by editing the registry

- a typical administrator has 2 logon sessions and only the limited session will be able to add/enumerate/remove a mapped drive
- you can use the `EnableLinkedConnections` policy to control whether the admin token will see the same drives as the limited token
- a WinRM process, will never see a persisted mapped drive as the automatic mapping only occurs with certain logon types
- when using credentials, use the Credential Manager to save the credential for the remote host
- when using the `/user:` parameter of `net use`, the drive will fail to map on the next logon, use the Credential Manager instead
- and lastly, use the `win_mapped_drive` and `win_credential` modules with Ansible to easily manage these resources