How We Found a PATH Vulnerability in Windows | ExpressVPN Blog

By The PATH to NT Authority/System

Archived: 2026-04-02 10:51:33 UTC

Editor's note: This post is part of our series for cybersecurity professionals and hobbyists, written by ExpressVPN's cybersecurity team.

In one of our regular security audits of ExpressVPN applications, we surfaced an interesting vulnerability. It wasn't found within our own codebase but stemmed from the use of .NET Core itself which, in the worst case, could result in privilege escalation on the Windows platform. However, the vulnerability could only be exploited if one of the controversial security boundaries was violated: the writable directory in PATH.

In this article, we will explain in detail the PATH environment variable and the security implications of having it misconfigured, and shed light on some of our findings in this space.

This PATH vulnerability issue was found during an audit of a closed beta version of our product for Windows not available to consumers. We remediated the issue after we discovered it and disclosed the discovery to Microsoft.

What is the PATH environment variable?

When a user tries to run a program from the command line interface (CLI), Windows needs to know the true location of the program in order to execute it. In our example below, we attempt to run ping. Windows doesn't immediately know where ping is located, but it still manages to find it.

```
C:\Users\User>ping

Usage: ping [-t] [-a] [-n count] [-l size] [-f] [-i TTL] [-v TOS]
            [-r count] [-s count] [[-j host-list] | [-k host-list]]
            [-w timeout] [-R] [-S srcaddr] [-c compartment] [-p]
            [-4] [-6] target_name
```

Internally, Windows looks up the location through the PATH environment variable. The PATH environment is a semi-colon (;) delimited list of directories, like the one below.

```
C:Windowssystem32;C:Windows;C:WindowsSystem32Wbem;C:WindowsSystem32WindowsPowerShellv1.0;C:WindowsSystem32OpenSSH;C:Pr
FilesMicrosoft SQL Server130ToolsBinn;C:Program
Filesdotnet;C:UsersUserAppDataLocalMicrosoftWindowsApps;;C:UsersUserAppDataLocalProgramsMicrosoft VS
Codebin;C:UsersUser.dotnettools
```

We note that PATH encompasses both the PATH environment variable for the system and the user, though for simplicity's sake the rest of the article will focus on the system's PATH environment variable. In our example, Windows searches each of the paths down the list for executables with the basename of ping, and executes the first one that matches (case insensitive). Here, the executable was found in the first entry of PATH, C:Windowssystem32, and C:WindowsSystem32PING.EXE is executed.

Time o...	Process Name	PID	Operation	Path
3:18:27...	cmd.exe	3904	CreateFile	C:\Users\User
3:18:27...	cmd.exe	3904	CreateFile	C:\Windows\System32
3:18:27...	cmd.exe	3904	CreateFile	C:\Windows\System32
3:18:27...	cmd.exe	3904	CreateFile	C:\Windows\System32
3:18:27...	cmd.exe	3904	CreateFile	C:\Users\User
3:18:27...	cmd.exe	3904	CreateFile	C:\Windows\System32\PING.EXE
3:18:27...	cmd.exe	3904	CreateFile	C:\Windows\apppatch\sysmain.sdb

One can think of a PATH as a list of trusted places in which Windows can look up programs to execute. This isn't just limited to the command line; Windows itself relies on the PATH environment variable to load dependencies, particularly dynamic link libraries (DLLs). If a certain DLL cannot be found, the PATH can be traversed to find the DLL.

PATH: An unmonitored attack surface

Needless to say, strict rules should be enforced on the paths in PATH for them to be considered trustworthy. In particular, **if the contents of any of the directories in PATH can be modified by any unprivileged local/remote user, then all bets are off**. Unfortunately, any administrator can haphazardly add directories into the PATH variable with no security checks performed by the operating system whatsoever. This can happen for a number of reasons:

1. **User error:** An administrator wishes to invoke a certain program just by its base name on the command line and adds the directory where the program resides without first checking the permissions on the directory.
2. **Vulnerable installation:** During the installation process, a benign program adds a directory with incorrect access controls to the PATH environment variable without locking down the permissions on that directory. Security researchers have surfaced vulnerabilities of this type in recent years, for example CVE-2020-12510.

There is much confusion on whether having a world-writable directory in PATH is a security risk. We definitively prove that it is a security risk with attack scenarios described below.

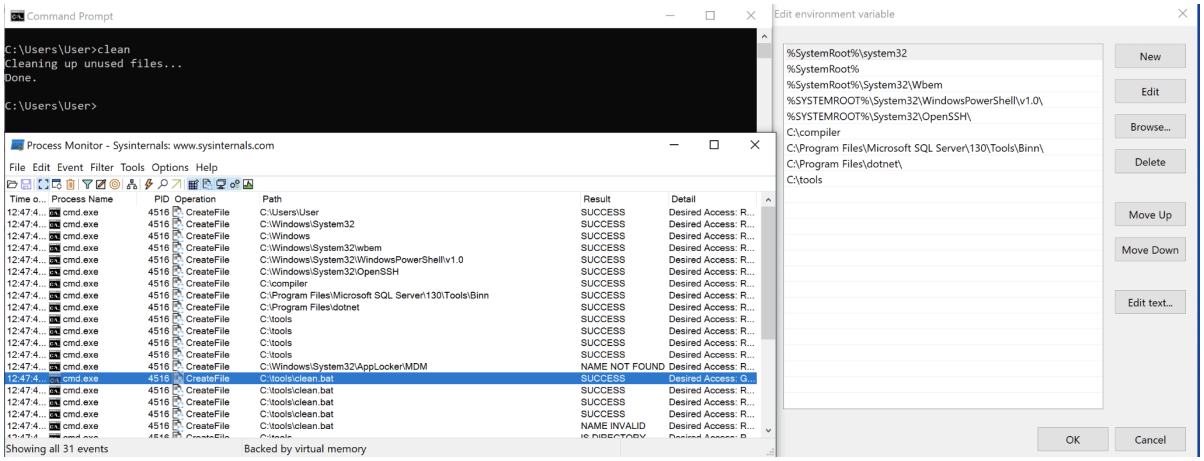
We present a roadmap to escalate privileges on the local machine, given a writable directory in PATH. Specifically, in the following attack scenarios, we assume the following are true:

1. The victim is running Windows

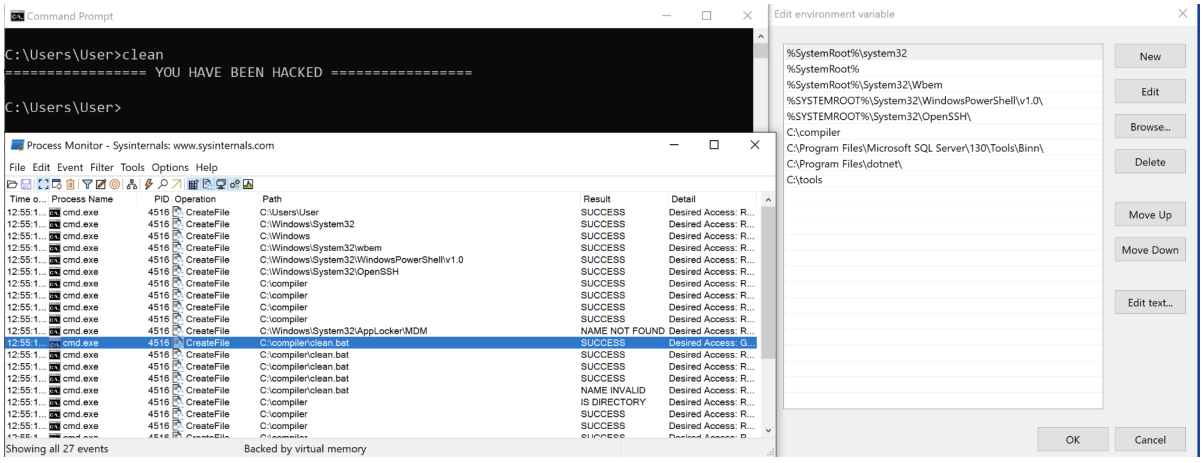
2. A directory listed in PATH is writable by any user
3. The attacker has the ability to write files to the directory in 2)

Scenario 1: PATH interception—search order hijacking

This exploitation scenario comes as a natural extension of how the PATH variable is used. Suppose an administrator regularly uses a custom program (clean.bat) to do administrative operations. He adds the directory it resides in, C:tools, to the system PATH variable for ease of invocation during his normal duties. The program runs when he invokes clean, and he happily goes off to perform his other duties.



Unbeknownst to him, the folder C:compiler in the PATH variable is world-writable, and its order in the PATH is much earlier than the program's true location. A malicious user notices and writes a script with the same name into the C:compiler folder. The next time the administrator runs clean from the command line, the clean program in C:compiler is executed:



From there, the administrator account is compromised—all it takes is to bypass User Account Control (UAC), which isn't a security guarantee and is trivially bypassable.

This technique is known as **search order hijacking**, where a malicious binary found earlier in the PATH variable executes in place of the expected benign executable. A surreptitious attacker can ensure that the original program is called after the payload is executed, meaning that the administrator would be none the wiser.

Note that this doesn't only apply to commands that are run by the administrator, but also by scripts and programs that invoke the shell to run a command. If any program or script isn't careful about how they are invoking other programs, then this vulnerability can be triggered.

During internal testing of a pre-release build of the Windows ExpressVPN application, we found such invocations which could be exploited when a directory in PATH is writable. During installation, the program needed to find the installed versions of the .NET runtimes and install the missing runtimes. If we look at the documentation, the "*recommended*" way is to invoke the dotnet executable, which programmatically would look like this:

```
var process = new Process
{
    StartInfo = new ProcessStartInfo("dotnet", "--list-runtimes")
    {
        UseShellExecute = false,
        RedirectStandardOutput = true,
        RedirectStandardError = true
    }
};
process.Start();
process.WaitForExit();
return process.StandardOutput.ReadToEnd();
```

In this case, because dotnet is found through the PATH environment variable, the search order can be hijacked and the malicious binary can run in its place. Our developers quickly triaged and remediated the issue by only loading dotnet from trusted installation paths, preventing such exploit techniques from working even when there is a writable directory in the PATH environment variable.

Scenario 2: DLL search path hijack on an external privileged application

Dynamic-Link Libraries (DLLs) are dependencies of an application packaged into their own binary file. This allows code to be shared without having multiple instances of the same code spread across files. When an application needs to invoke a method from one of these libraries, they load the DLL into memory and execute the relevant function in the DLL. Because code is executed from the DLL, it must be trusted and cannot be arbitrarily loaded.

When a Windows application loads, some DLLs may be missing at the start. In this case, the application will try to look for missing DLLs that it needs in the following manner:

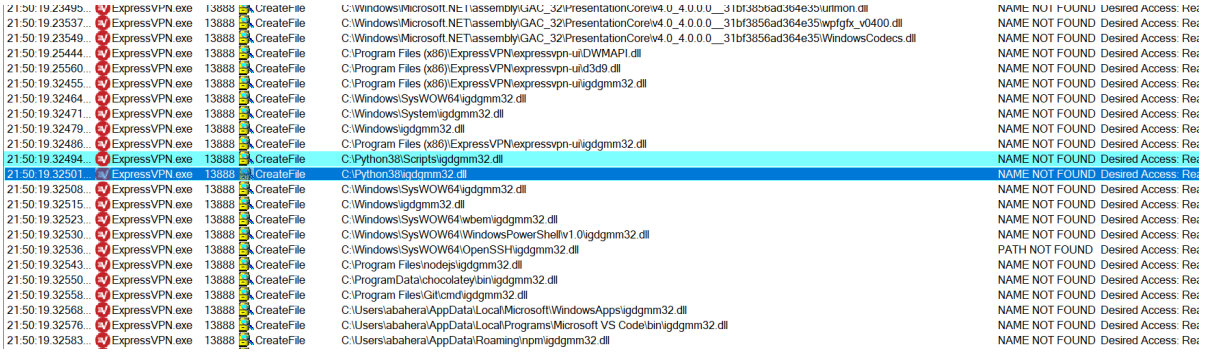
- by first looking at the directory it was loaded from (where the application resides)
- looking at the current directory
- the system directory
- the Windows directory
- searching in the directories listed in the PATH environment variable

In ordinary circumstances, this behavior is warranted. Developers are able to prevent their application from loading outside of the desired directories by making sure that they include all dependencies which the program needs in the folder where the program resides.

However, this isn't always the case, if Windows doesn't find the DLL it needs, it can fall back to loading from the directories of the PATH environment variable. This can be hijacked by a malicious user with write access to a directory in PATH, using a technique known as **Phantom DLL Hijacking**. We list two case studies below where privileged applications (e.g., drivers, services) show this behavior.

The curious case of the driver

In July 2020, we received a submission from a BugCrowd researcher claiming to have found a privilege escalation for our Windows application. The researcher provided the following proof that ExpressVPN was loading the igdgmm32.dll from a directory in the PATH environment variable.



21:50:19.23495...	ExpressVPN.exe	13888	CreateFile	C:\Windows\Microsoft.NET\assembly\GAC_32\PresentationCore\v4.0_4.0.0.0__31bf3856ad364e35\unimon.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.23537...	ExpressVPN.exe	13888	CreateFile	C:\Windows\Microsoft.NET\assembly\GAC_32\PresentationCore\v4.0_4.0.0.0__31bf3856ad364e35\wpfgfx_v0400.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.23549...	ExpressVPN.exe	13888	CreateFile	C:\Windows\Microsoft.NET\assembly\GAC_32\PresentationCore\v4.0_4.0.0.0__31bf3856ad364e35\WindowsCodecs.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.25444...	ExpressVPN.exe	13888	CreateFile	C:\Program Files (x86)\ExpressVPN\expressvpn-ui\DWMAPI.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.25560...	ExpressVPN.exe	13888	CreateFile	C:\Program Files (x86)\ExpressVPN\expressvpn-ui\d3d9.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32455...	ExpressVPN.exe	13888	CreateFile	C:\Program Files (x86)\ExpressVPN\expressvpn-ui\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32464...	ExpressVPN.exe	13888	CreateFile	C:\Windows\SysWOW64\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32471...	ExpressVPN.exe	13888	CreateFile	C:\Windows\System\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32479...	ExpressVPN.exe	13888	CreateFile	C:\Windows\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32486...	ExpressVPN.exe	13888	CreateFile	C:\Program Files (x86)\ExpressVPN\expressvpn-ui\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32494...	ExpressVPN.exe	13888	CreateFile	C:\Python38\Scripts\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32501...	ExpressVPN.exe	13888	CreateFile	C:\Python38\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32508...	ExpressVPN.exe	13888	CreateFile	C:\Windows\SysWOW64\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32515...	ExpressVPN.exe	13888	CreateFile	C:\Windows\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32523...	ExpressVPN.exe	13888	CreateFile	C:\Windows\SysWOW64\wbem\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32530...	ExpressVPN.exe	13888	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32536...	ExpressVPN.exe	13888	CreateFile	C:\Windows\SysWOW64\OpenSSH\igdgmm32.dll	PATH NOT FOUND Desired Access: Rea
21:50:19.32543...	ExpressVPN.exe	13888	CreateFile	C:\Program Files\nodejs\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32550...	ExpressVPN.exe	13888	CreateFile	C:\ProgramData\chocolatey\bin\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32558...	ExpressVPN.exe	13888	CreateFile	C:\Program Files\Git\cmd\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32568...	ExpressVPN.exe	13888	CreateFile	C:\Users\ababera\AppData\Local\Microsoft\WindowsApps\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32576...	ExpressVPN.exe	13888	CreateFile	C:\Users\ababera\AppData\Local\Programs\Microsoft VS Code\bin\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea
21:50:19.32583...	ExpressVPN.exe	13888	CreateFile	C:\Users\ababera\AppData\Roaming\npm\igdgmm32.dll	NAME NOT FOUND Desired Access: Rea

The initial triage conducted by Bugcrowd was unable to reproduce the finding. Our internal triage team then took a look to ensure we do not miss any valid findings. We first studied the impact of the issue; **ExpressVPN.exe never runs in a privileged context and only as the current user**, so the privilege escalation would be lateral in the worst case (from one user to another user). However, we do not rule out the possibility of a path to successful privilege escalation; one could attempt to compromise the administrator running ExpressVPN.exe from another non-privileged user account on the same machine.

Next, we looked at the prerequisites for exploitation to be successful. The installation process of ExpressVPN ensures that the correct access controls are set for the folders that are created. **This means that the folders where ExpressVPN.exe resides are not writable by non-privileged users**. The proof-of-concept the researcher provided loads from C:Python38, a directory unrelated to ExpressVPN and which was already world-writable to begin with, likely due to a misconfiguration on the Administrator's part. This means most of our users are not impacted unless they have a world writable directory to begin with.

We then dug deeper into the underlying issue. On our virtual machine testbeds, we similarly could not reproduce the finding, but we were convinced that it was possible thanks to the Procmon logs provided by the researcher. We investigated the DLL itself, and found that igdgmm32.dll is part of the Intel HD Graphics Drivers package. From there, we quickly isolated the issue and found this article on DLL Hijacking. It turns out that the driver is loaded as part of the Windows Presentation Foundation (WPF) UI framework (a framework used to build Windows UI applications), and this only occurs when the system chipset used belongs to Intel. Indeed, this was the missing dependency that was causing the library to load from the PATH environment variable.

Unfortunately, in this case, because **the underlying problem impacts any application that uses the Windows Presentation Foundation UI framework**, there's very little that can be done on our end short of the mitigations we already have in place. A complete fix would require the vendor to patch the issue.

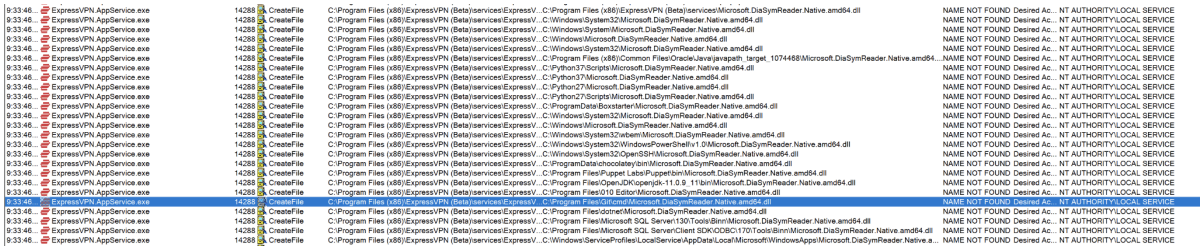
We contacted both Microsoft and Intel on the issue, crediting the original researcher for the discovery. Microsoft claimed it wasn't under their purview and nudged us towards notifying Intel. Intel investigated the issue and found that the issue raised in the report was present in an older version of the driver (26.20.100.7262), and the latest version at the time (27.20.100.8476) no longer had the issue.

Even without a vendor patch, it is important to emphasize that this can only occur if a directory in PATH is writable, which by default isn't the case, so we reiterate the importance of checking your PATH.

O dependency, where art thou?

In our second case study, we examine a vulnerability found during an audit of our next-generation Windows application (pre-release). During our testing, we observed the dependencies loaded via Procmon; one dependency caught our attention -

Microsoft.DiaSymReader.Native.amd64.dll. We noticed it was loaded every time a service belonging to our application started or restarted.



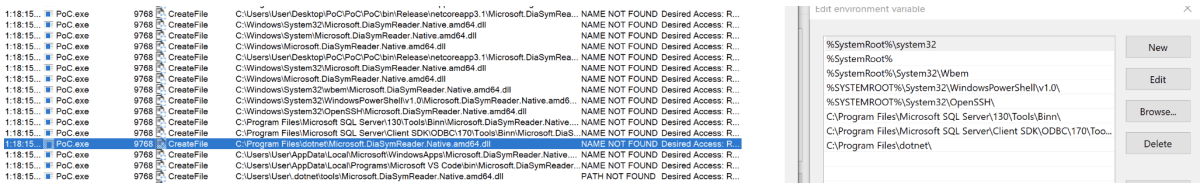
Much like the DLL hijacking above, the DLL can be hijacked and loaded from a directory in the PATH environment variable. The AppService was running as NT AUTHORITYLOCAL SERVICE, which by default isn't privileged. However, because of improper permission boundaries in Windows, if we can get code execution on the AppService via planting a malicious DLL in a writable PATH directory, we can leverage one of the existing bypasses to escalate privileges to NT AUTHORITYSYSTEM. This is unfortunate because while NT AUTHORITYLOCAL SERVICE was designed to have less privileges, due to the complexity of the Windows operating system, not all privilege escalation paths are covered. Those interested can read more about the privilege escalations possible here.

Our development team investigated for some time, but there was no reference to this library anywhere in our codebase. Somehow or somewhere, *something* was loading this DLL, and we had to get to the bottom of it to identify the root cause and protect our users.

We then did something quite extreme. From the original codebase, they took out large chunks of the code, recompiled, then ran the program to see if the DLL was still loaded. Through this, we were effectively able to "binary search" through the codebase to identify the root cause. As it turns out, Microsoft.DiaSymReader.Native.amd64.dll was loaded because an unhandled exception was thrown in the program. The offensive security team quickly created a proof of concept below.

```
namespace PoC
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var a = args[100]; // triggers IndexOutOfRangeException
        }
    }
}
```

We executed the PoC, and lo and behold, Microsoft.DiaSymReader.Native.amd64.dll was loaded!



We reported the issue to Microsoft on January 5, 2021. On January 21, Microsoft determined that the vulnerability did not meet the bar for servicing because *"Adding folders to the PATH variable requires administrator rights, and all default folders that are included in the PATH variable require administrator permissions to modify."* This is consistent with how they handled similar vulnerabilities in the past (more information on this in Scenario 3).

The development team, upon realizing the issue, remediated it immediately by packaging the missing DLLs with the installation. The files Microsoft.DiaSymReader.Native.amd64.dll for the 64-bit version and Microsoft.DiaSymReader.Native.x86.dll for the 32-bit version are installed in the application folder where the service applications reside so the files will no longer be loaded from PATH. **This means that our application remains safe,** regardless of Microsoft's decision to publish a patch for this issue.

Scenario 3: DLL search path hijack on default Windows applications

Even if you choose to install nothing on your machine, some in-built Windows applications may themselves be prone to DLL Hijacking. Security researchers have extensively studied this exploitation technique and published detailed findings on different scenarios in which this can be exploited. Windows applications tend to run as privileged services or applications, so an exploitable DLL Hijack would allow a user to run code as that application to escalate privileges. The prerequisite, as always, is that a directory in PATH is writable for our less-privileged user.

What is particularly striking is not the existence of such issues, but the vendor's stance towards fixing this issue. There are two widely known targets where this could be exploited, one that works for Windows 8 and below and another that works for Windows 10. For both of these issues, Microsoft deemed that they did not meet the bar for servicing. We note that the former exploit technique was mitigated silently in 2013 with the release of Windows 8.1, while no news of patches for the latter. There is no defense in depth: Once a machine has a PATH directory without proper access controls, a malicious process/user can easily escalate privileges and perform any kind of malicious activity as it pleases.

What does this mean for the end user?

As of April 2018, Microsoft has made it clear that vulnerabilities that require a directory in PATH to be writable are not vulnerabilities since the directories there should never be world-writable, and if they were, then it is the fault of the administrator or application that made that change. As seen in Scenarios 2 and 3, exploitation techniques that leverage the writable PATH directories are unlikely to be mitigated by official patches, so it is up to end users to enforce that all directories in PATH have the proper access controls in place.

Thankfully, such vulnerabilities aren't common. Users rarely change the system PATH environment variable, and it is even rarer for them to specify a world-writable directory there, though that is definitely possible. It is more likely that an installation process introduces a vulnerability. However, from our quick investigation of approximately 100 of the top applications for Windows from various download sites, none of the installations introduced this vulnerability.

Here at ExpressVPN, we put in place in-depth defensive measures to ensure that even if the PATH contains a writable directory, our applications never load code from such directories. We do so by ensuring that executables are run from trusted paths, along with mitigating DLL planting vulnerabilities when necessary.

Nevertheless, one can never be too careful. To that end, we have created a simple PowerShell script which you can use to audit your system. As with any script you find on the internet, read through carefully to make sure you understand what it is doing.

You can run the script as a low privilege user to see if any of the directories in the system PATH are writable, and the script outputs the writable directories into the command line, as below:

```
PS C:\Users\lowpriv\Desktop> .\has_writable_PATH.ps1
'C:\compiler' is writable!
Press Enter to continue...:
```

You can secure yourself by removing such directories from your system PATH, or by appropriately changing the permissions of the directories listed. Once that's done, you can re-run the script and check the results to make sure your system is truly hardened!

```
PS C:\Users\lowpriv\Desktop> .\has_writable_PATH.ps1
No writable directories in system PATH found!
Press Enter to continue...:
```

It's important for users to stay vigilant in protecting their privacy and security. We believe in empowering users and enabling them to protect themselves by arming them with the tools and technologies to do so. Our discussion on PATH vulnerabilities should dispel some myths and we hope that our script has helped our readers identify vulnerabilities on their machines before they can be exploited by malicious actors.

```
function Test-Administrator
{
    $domain, $userToFind = [String]([Security.Principal.WindowsIdentity]::GetCurrent().Name) -Split ''
    $administratorsAccount = Get-WmiObject Win32_Group -filter "LocalAccount=True AND SID='S-1-5-32-544'"
    $administratorQuery = "GroupComponent = `"Win32_Group.Domain='" + $administratorsAccount.Domain +
"',NAME='" + $administratorsAccount.Name + "'`""
    $user = Get-WmiObject Win32_GroupUser -filter $administratorQuery | select PartComponent |where {$_ -match
"$domain"}  |where {$_ -match "$userToFind"}
    $user
}
if (Test-Administrator) {
    Write-Host "Please run as a low privilege user for accurate results."
    Exit 1
}
function Is-Writable-Folder {
    param (
        [parameter(Mandatory=$True)]
        [string]$Folder
    )
    if (Test-Path "$Folder" -IsValid) {
        # folder exists
        $TestFile = "__test_PATH_$(get-date -f MM-dd-yyyy_HH_mm_ss).tmp"
        $TestFilePath = Join-Path -Path $Folder -ChildPath $TestFile
        Try {
            [io.file]::OpenWrite($TestFilePath).close()
            [io.file]::Delete($TestFilePath)
            return $true
        } Catch {
            return $false
        }
    } else {
        return $false
    }
}
$AllPaths = (Get-ItemProperty -Path 'Registry::HKEY_LOCAL_MACHINESystemCurrentControlSetControlSession
ManagerEnvironment' -Name PATH).Path.Split(";")
$WritablePaths = @()
Foreach ($path in $AllPaths)
{
    Try {
        $path = $path.Trim()
```

```
        if ($path.Length -gt 0) {
           if (Is-Writable-Folder($path)) {
            $WritablePaths += $path
          }
        }
    } Catch {
        Write-Host "Error processing: '$path' - not a valid directory"
    }
}
If ($WritablePaths.count -eq 0) {
    Write-Host "No writable directories in system PATH found!"
} Else {
    Foreach ($path in $WritablePaths) {
        Write-Host "'$path' is writable!"
    }
}
Pause
```

Source: https://www.expressvpn.com/blog/cybersecurity-lessons-a-path-vulnerability-in-windows/