

SrClient DLL Hijacking: a Windows Server 2012 0-day that won't be patched

 blog.vonahi.io/srclient-dll-hijacking

February 19, 2021



19 February 2021 / [privilege escalation](#)

 SrClient DLL Hijacking: a Windows Server 2012 0-day that won't be patched

I recently discovered that all versions of Windows Server 2012 (but not Server 2012 R2) are affected by a DLL hijacking vulnerability that can be exploited for privilege escalation. Moreover, the flaw can be triggered by a regular user and does not require a system reboot. Sounds like a pretty big deal, right? Well, not according to Microsoft, unfortunately. The vulnerability relies on `%PATH%` containing at least one insecurely configured directory, and Microsoft does not consider bugs of this category to be security vulnerabilities worthy of a fix. However, if your company is running Windows Server 2012, there is a decent chance this vulnerability can allow regular users (or attackers with access as a regular user) to pwn your server or domain controller. Let me show you how.

0x00 A Refresher on DLLs and %PATH%

If you are unfamiliar with DLL hijacking, feel free to check out [this blog post](#) that contains a general introduction to this subject. A brief summary of key concepts follows below.

DLLs

Microsoft defines a DLL as:

a library that contains code and data that can be used by more than one program at the same time.

DLLs are very similar to EXE files, but they can only be executed after being called by an EXE.

DLL Search Order

Microsoft says [this](#) about the DLL Search Order:

when an application loads a DLL without specifying a fully qualified path, Windows attempts to locate the DLL by searching a well-defined set of directories in an order known as **DLL search order**.

Starting with Windows XP SP2, the default DLL search order on Windows systems is something like this:

1. The directory from which the application loaded.
2. The system directories. On modern 64-bit systems these are C:\Windows\System32 (64-bit programs and libraries - yes you are reading that right, the names are counterintuitive) and C:\Windows\SysWOW64 (32-bit programs and libraries). SysWOW64 is logically absent on 32-bit systems, where C:\Windows\System32 coexists with C:\Windows\System (16-bit programs and libraries).
3. The Windows directory (C:\Windows)
4. The current directory.
5. The directories that are listed in the `%PATH%` environment variable.

PATH

PATH is an environment variable in Windows as well as Unix-like operating systems including Linux and MacOS. Basically, PATH is a special kind of variable that specifies a set of directories where executable programs are located. In Windows, this variable is referenced as `%PATH%`. For more info, see the [Wikipedia entry](#).

DLL hijacking and %PATH%

On a clean installation of any modern Windows system, `%PATH%` does not contain directories with weak permissions that would allow for the attack described in this article. However, many third-party applications add directories to `%PATH%` during installation and those directories aren't always securely configured. As a result, it is not uncommon in corporate environments to find Windows systems that allow one or more regular users to write arbitrary data to `%PATH%` directories. If this is the case on a Windows Server 2012 system, it could be child's play for an attacker with the privileges of one such regular user to escalate privileges to `NT AUTHORITY\SYSTEM` via `SrClient.dll` hijacking.

0x01 Identifying the Vulnerability

In this article I'm standing on the shoulders of giants. Well, I don't know how tall Clément Labro (@itm4man) is, but I found this vulnerability as a result of their discovery of the NetMan DLL Hijacking vulnerability that affects all editions of Windows Server, from 2008R2 to 2019. I recently came across their excellent writeup of this issue, and decided to spin up a Windows Server VM to try and replicate their findings. I have multiple Windows VMs set up for research purposes, and it was pure chance that I picked a Server 2012 system.

After booting up, I launched Process Monitor (procmon64.exe) and added a few filters to have it display any and all failed attempts by running processes to load a DLL or EXE file from `C:\Windows\System32\WindowsPowerShell\v1.0\`, which is part of %PATH% by default. Events that match these filters would most likely indicate that Windows was trying to load a non-existing resource by relying on the DLL search order, and would therefore represent potential DLL Hijacking / Binary planting vulnerabilities.

Initially the events field remained empty, and when I finally started getting a few results, none of them were for the `wlanapi.dll` or `wlanhlp.dll` resources, which would be evidence of the NetMan DLL vulnerability. So far this was expected, since that vulnerability is only triggered under specific circumstances. I therefore started looking into Clément Labro's exploit to trigger the flaw. After a while I glanced at my VM again, and noticed something interesting: the process `TiWorker.exe` had tried to load a resource called `SrClient.dll` from the aforementioned PowerShell directory. I inspected the event and noticed that the process was running as `NT AUTHORITY\SYSTEM`.

The screenshot shows the Windows Task Manager interface. In the background, the 'Processes' tab is active, displaying a list of running applications. The process 'TiWorker.exe' is selected and highlighted with a red rectangular box.

In the foreground, the 'Event Properties' dialog box is open, displaying details for the selected event. The 'Process' tab is selected, showing the following information:

- Name:** TiWorker.exe
- Version:** 6.2.9200.16384 (win8_rtm.120725-1247)
- Path:** C:\Windows\winsxs\amd64_microsoft-windows-servicingstack_31bf3856ad364e35_6... (partially obscured)
- Command Line:** C:\Windows\winsxs\amd64_microsoft-windows-servicingstack_31bf3856ad364e35_6... (partially obscured)
- PID:** 2008
- Parent PID:** 616
- Session ID:** 0
- User:** NT AUTHORITY\SYSTEM (highlighted with a red rectangular box)
- Auth ID:** 00000000:000003e7
- Started:** 2/11/2021 3:54:52 AM
- Ended:** (Running)

Moreover, the Stack Trace of the event included references to `rpcrt4.dll`, which Clément Labro mentioned as a sign that the event was likely triggered via RPC/COM and could therefore possibly be triggered by a regular user. In line with the NetMan DLL

hijacking write-up, I then launched a search query on my Windows 10 host for `SrClient.dll` , and I found it at `C:\Windows\System32\SrClient.dll`

```
C:\>dir C:\Windows\System32\srclient.dll
Volume in drive C is OS
Volume Serial Number is 1AE5-FAAB

Directory of C:\Windows\System32

12/07/2019  11:09 AM                74,752 srclient.dll
              1 File(s)                74,752 bytes
              0 Dir(s)  33,911,750,656 bytes free
```

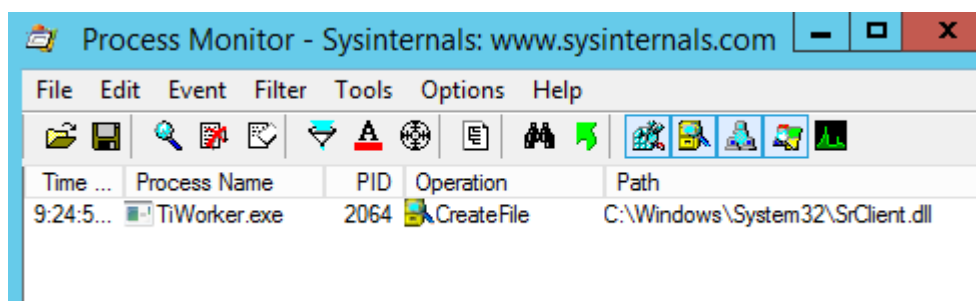
So far my findings were almost identical to those for the NetMan DLL hijacking vulnerability:

- A process on a Windows Server edition tried to load a non-existent DLL via the DLL Search Order
- The calling process was running as `NT AUTHORITY\SYSTEM`
- The event was likely triggered via RPC/COM
- The DLL did exist on Windows 10

At this point, I was starting to believe that I may have actually stumbled onto something big, but I tried to compose myself as I knew it might not be possible to actually trigger the vulnerability as a regular user. While I proceeded to look into this right away, I first want to address some discoveries I made later regarding the systems that are actually affected by this.

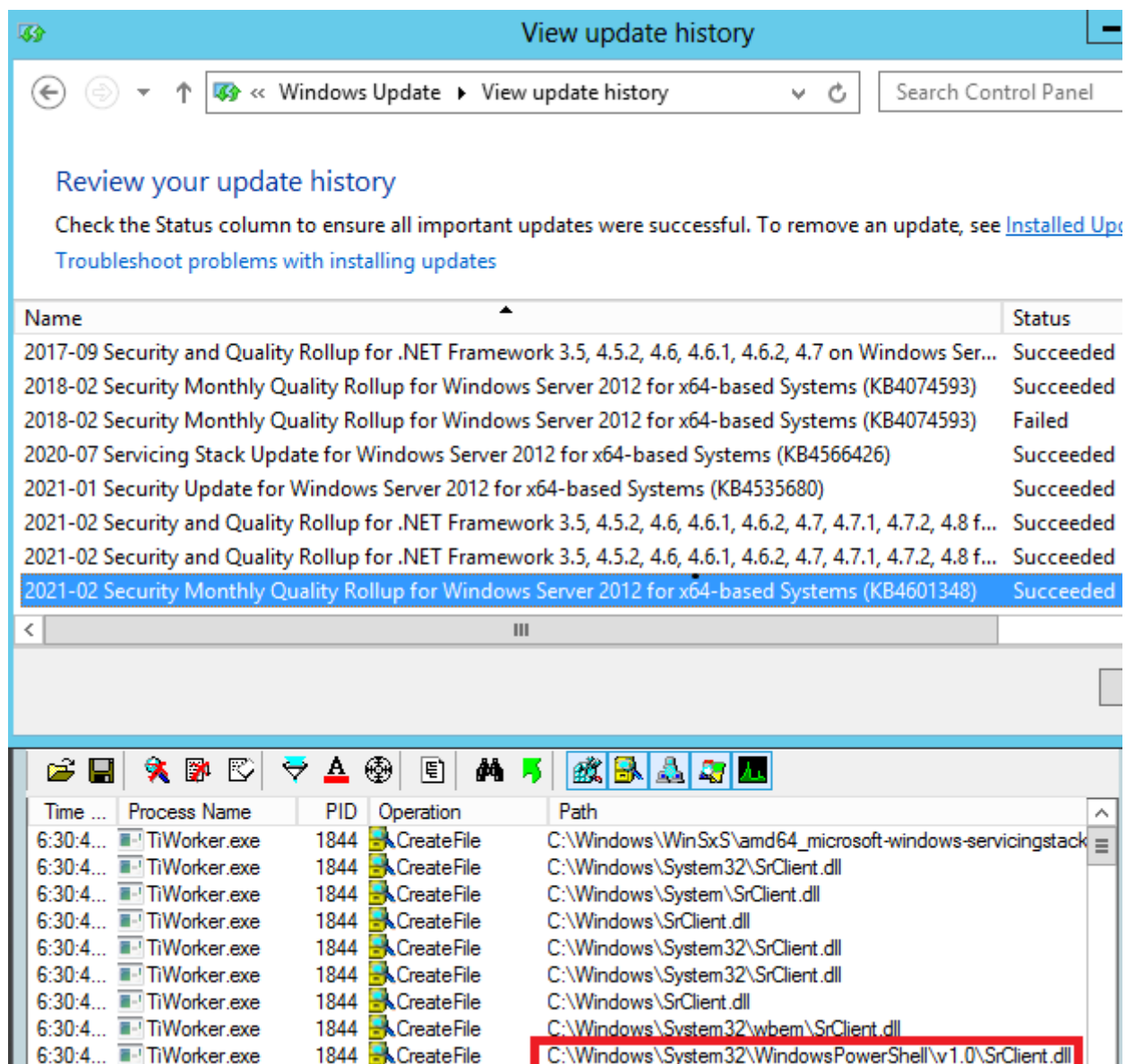
Checking for affected systems

When I tried to replicate my findings on other Windows Server versions, I discovered that none of them seemed vulnerable. On Windows Server 2016, 2019 and to my surprise even 2012R2, `SrClient.dll` does not exist and `TiWorker.exe` will try to load it, but only from `C:\Windows\System32\` , which is the correct path in Windows 10. Because the DLL Search Order is not used, DLL hijacking to achieve privilege escalation is out of the question.



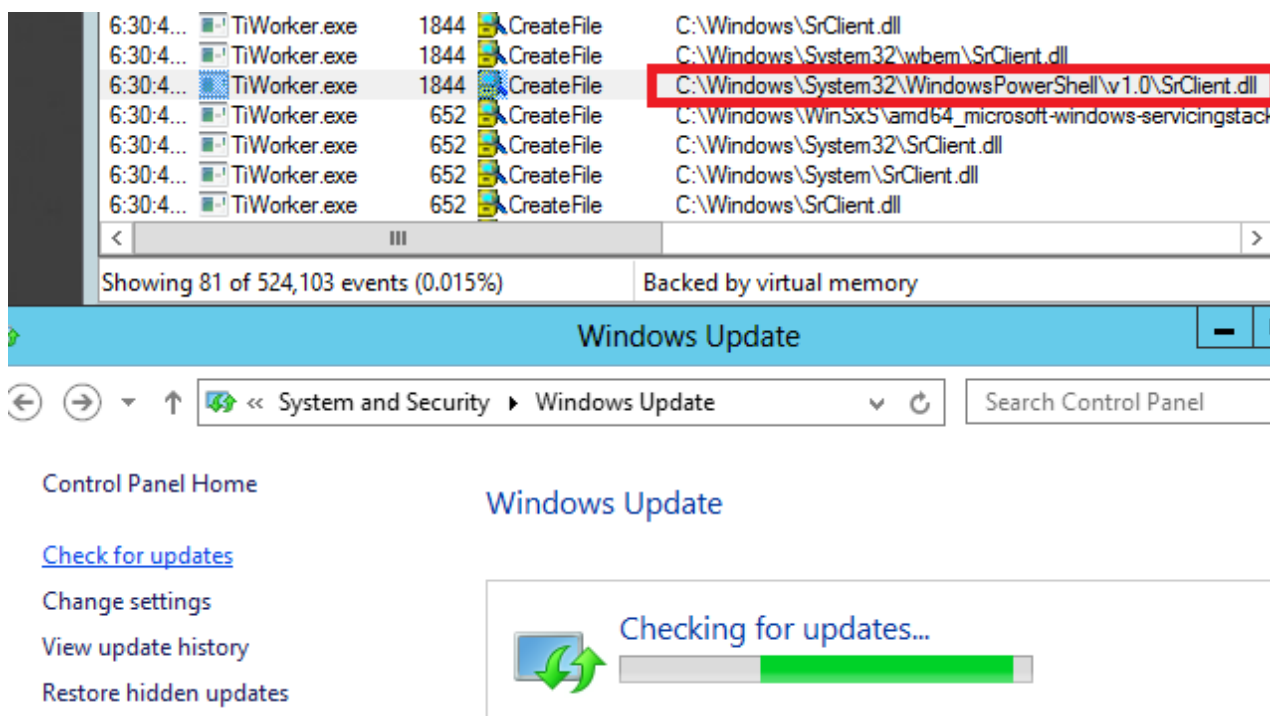
On Windows Server 2008R2 I wasn't able to trigger this event at all, so I assume that OS isn't vulnerable either. I don't have a Windows Server 2008 VM to test this on, but my guess is that it doesn't differ from 2008R2 when it comes to `SrClient.dll`.

Somewhat dismayed by these findings, and a little concerned about the fact that Windows Server 2012R2 didn't even seem vulnerable, I downloaded a fresh Windows Server 2012 ISO evaluation image from the [Microsoft Evaluation Center](#) and installed all possible updates on it, including this month's Patch Tuesday rollup (KB4601348). I then tried to trigger the vulnerability (which I had learned how to do, see below) and... it worked!



0x02 Triggering the Vulnerability

I started my search for a way to trigger this vulnerability at the source: `TiWorker.exe`. I probably should have been familiar with this process, but I wasn't. Fortunately, a quick web search [revealed](#) that it is part of the *Windows Module Installer Service*, the purpose of which is to download and install Windows Update packages. It resides in `C:\Windows\servicing`, just like its parent process `TrustedInstaller.exe`, which is also part of the *Windows Module Installer Service*. Because of the link with Windows Update, I decided to check if I could get `TiWorker.exe` to launch by checking for updates on my test system via the Control Panel. To my amazement, this worked right off the bat!



Of course, in order to be able to exploit this in a real-world scenario, it would be far better to pull the trigger from the command line, that is CMD.exe or PowerShell. My initial search directed me toward the latter, but fortunately [Alton](#) suggested I check out **WUAUCLT** (Windows Update Automatic Update Client) and its [commands](#), all of which can be run from CMD.exe. And sure enough, after some trial and error, I managed to trigger the vulnerability when running **WUAUCLT** with the one of the following commands:

- **/SelfUpdateManaged** - This launches the Windows Update window in the Control Panel and tells it to start checking for updates using Windows Server Update Services (WSUS).
- **/SelfUpdateUnManaged** - Similar to the one above, but it uses the Windows Update website instead of WSUS.
- **/DetectNow** - This will detect and download available updates in the background.

Of these possible triggers, only **WUAUCLT /DetectNow** is relatively stealthy because it will run in the background. The other two will launch the Windows Update UI, which a legitimate user would obviously notice. However, even in that scenario, users may not actually recognize this event as something malicious. Windows Update has a reputation of pushing updates in the absence of informed consent by users (mostly because the latter haven't properly configured it). As a result, some users would probably interpret the event as just another example of how capricious the update service is.

By now, I felt like jumping out of my chair with joy, but I tried to compose myself as I still needed to verify a few things, namely:

- Could a regular user trigger the vulnerability in the manner just described?
- Would this also work for a world-writable directory that I would add to %PATH%? (I couldn't think of a reason why it shouldn't, but you never know).

- If the previous conditions were met, could I actually get a reliable reverse shell by writing a payload named `SrClient.dll` to my world-writable `%PATH%` directory and getting `TiWorker.exe` to load it?

In order to check all of this, I started by creating a regular user account `wynter` with only the most basic privileges:

```
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

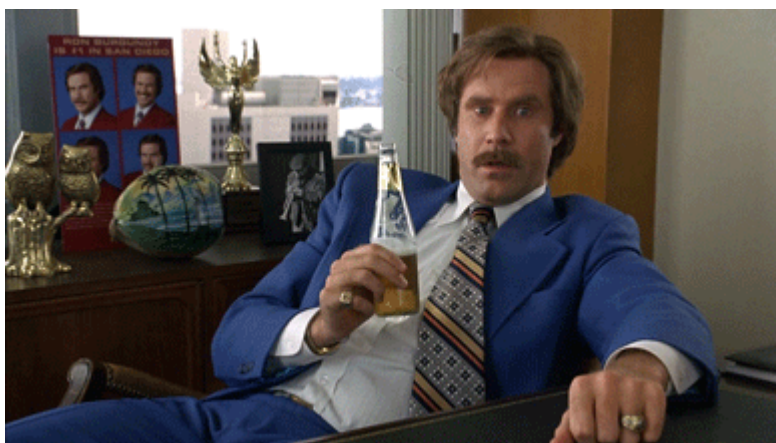
PS C:\Users\wynter> whoami /priv

PRIVILEGES INFORMATION
-----
Privilege Name            Description                State
-----
SeChangeNotifyPrivilege   Bypass traverse checking   Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set Disabled
PS C:\Users\wynter>
```

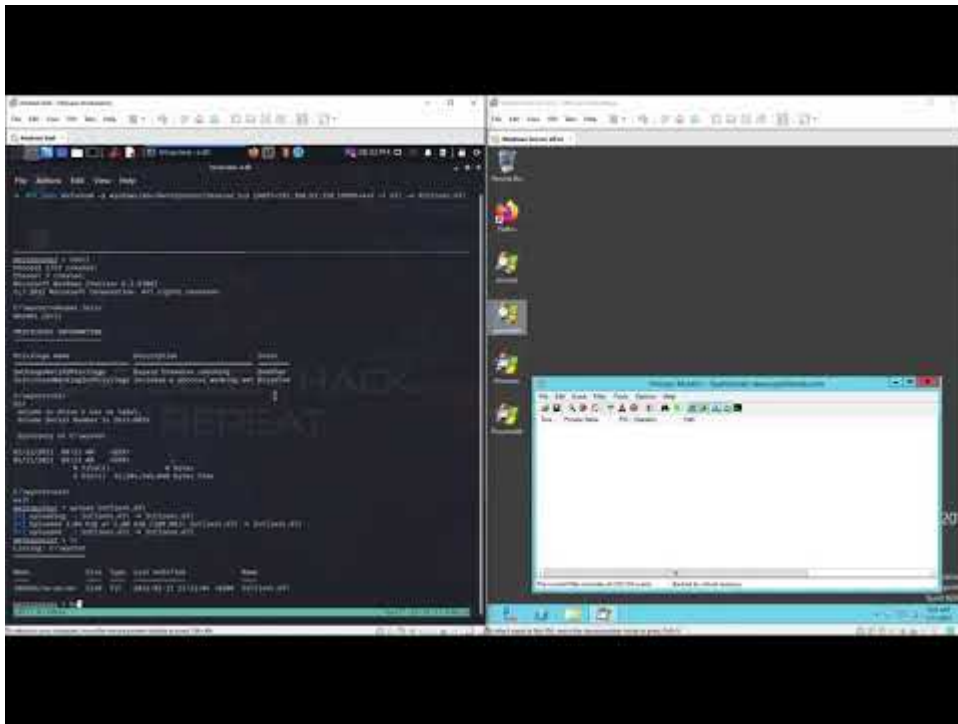
As the above image shows, `wynter` didn't even have the `SeShutdownPrivilege` that is required to shutdown or reboot the OS. Next, I created a directory at `C:\wynter` and added it to `%PATH%`. Finally, I used MsfVenom to generate two reverse Meterpreter shells (x64):

- An EXE payload that I executed as `wynter` on the target in order to get a remote session with limited privileges on the Server 2012 VM
- A DLL payload called `SrClient.dll`.

After establishing my unprivileged Meterpreter session, I used it to upload `SrClient.dll` to `C:\wynter\`. Then I dropped into a CMD shell and ran `wuauc1t /SelfUpdateManaged`. The result? Well...



Video - Manual Exploitation



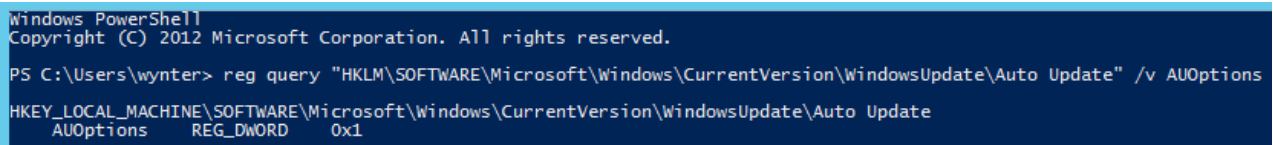
Watch Video At: <https://youtu.be/JpZQcemvbv4>

0x03 Writing a Metasploit Module

After figuring out how to exploit this issue manually, I decided to take it to the next level and write a Metasploit module for it. This was my first local exploit module, and it was quite a pain to write, especially because the exploit proved to be less reliable than I had hoped. The next section outlines the limitations I discovered while working on this module. The below video shows the full attack using a draft of the module for which I have opened a ***pull request***. The final version of the module will probably turn out a little different.

Video - Metasploit Exploitation

```
reg query  
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate\Auto  
Update" /v AUOptions
```



```
Windows PowerShell  
Copyright (C) 2012 Microsoft Corporation. All rights reserved.  
  
PS C:\Users\wynter> reg query "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate\Auto Update" /v AUOptions  
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate\Auto Update  
AUOptions REG_DWORD 0x1
```

As the above image shows, the value is returned in the following format:

```
AUOptions REG_DWORD 0x<value>
```

There are four possible values:

- **1** - Never check for updates (doesn't work with `/DetectNow`)
- **2** - Notify before downloading and installing any updates
- **3** - (Default setting) Download the updates automatically and notify when they are ready to be installed
- **4** - Automatically download updates and install them according to a specified schedule.

On the other hand, if setting 3 or 4 is enabled on the target system, an attacker with a lot of time on their hands would not have to trigger the exploit themselves at all. Instead, they could simply write their payload to a vulnerable directory, and wait for Windows Update to launch and trigger the payload for them.

x64 vs x86

So far I have only been able to test this attack on a x64 Windows Server 2012 system and on my test system, exploitation seemed to require a 64-bit (x64) compatible payload.

Mitigation

I almost forgot to address what you can actually do to prevent this attack in your environment. As mentioned, Microsoft won't release a patch, so keeping your systems updated (which is generally a great idea), will not save you here. What will do that, is making sure that `%PATH%` does not include directories with weak permissions. In addition, you can consider the following additional best practices:

- Limit the number of local user accounts on your servers.
- Ensure that each account has a unique, complex password.
- Install an advanced endpoint security solution on all your servers to increase the likelihood of attacks of this kind getting blocked or at least detected.

Finally, you could of course also consider upgrading to a newer edition of Windows Server. This may seem drastic, but keep in mind that Windows Server 2012 will reach end-of-life in 2023 and it's generally a good idea to start planning your migration long in advance.

Acknowledgements

- As mentioned, none of this would have happened without the fantastic, accessible research done by Clément Labro (@itm4man) on the NetMan vulnerability.
- Thanks goes out to Trammie for the stunning image she designed for this post, and to Alton and the rest of the team at Vonahi Security for making it so easy to enjoy my work.
- Finally, I couldn't have done this without my family tolerating my many loud, unexpected outbursts along the lines of "WOW THIS IS INTERESTING!"; "WHY THE HELL ISN'T THIS WORKING!?" ; "WHY DOES THIS #\$\$@%\$ VM KEEPS FREEZING!?" and of course "WELL THAT ESCALATED QUICKLY!"