Bypassing UAC using App Paths

Published: 2017-03-14 · Archived: 2026-04-05 21:21:52 UTC

Over the past several months, I've taken an interest in Microsoft's User Account Control (UAC) feature in Windows. While Microsoft doesn't define UAC as a security boundary, bypassing this protection is still something attackers frequently need to do. The recent Vault7 leak confirms that bypassing UAC is operationally interesting, even to nation states, as several UAC bypasses/notes were detailed in the dump. On the purposefully public side, check out the UACME project by @hfiref0x for a great collection of existing techniques.

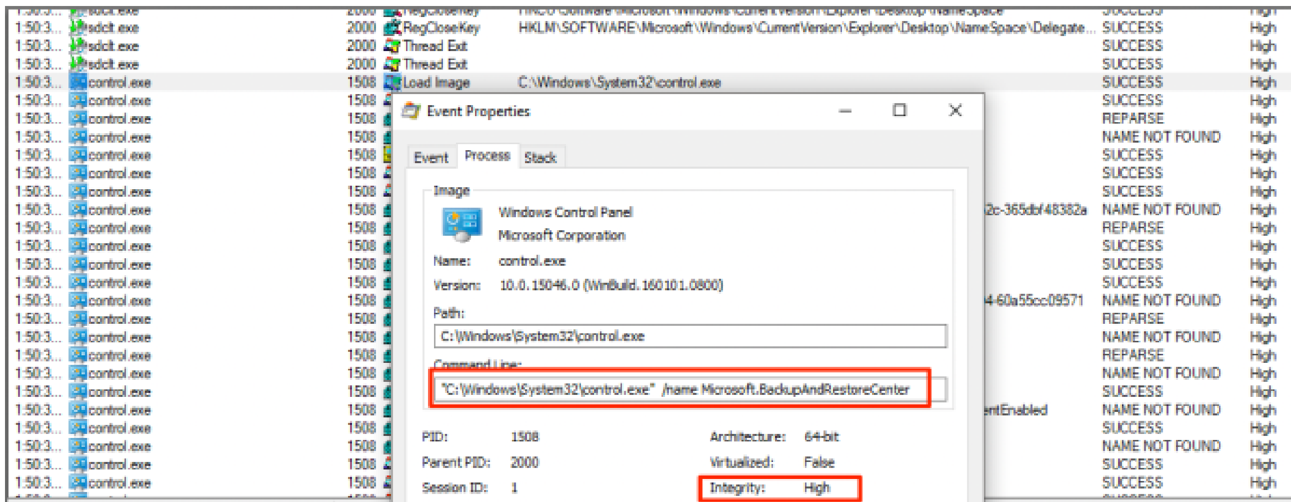
Microsoft seems to have a renewed interest in UAC, finally fixing many of the issues highlighted by publicly disclosed bypasses. While these fixes are only in newer versions of Windows, the active response from Microsoft drives me to continue searching for UAC bypasses. I've previously blogged about two different bypass techniques, and this post will highlight an alternative method that also doesn't rely on the IFileOperation/DLL hijacking approach. This technique works on Windows 10 build 15031, where the vast majority of public bypasses have been patched.

As some of you may know, there are some Microsoft signed binaries that auto-elevate due to their manifest. You can read more about these binaries and their manifests here. While searching for more of these auto-elevating binaries by using the SysInternals tool "sigcheck", I came across "sdclt.exe" and verified that it auto-elevates due to its manifest:

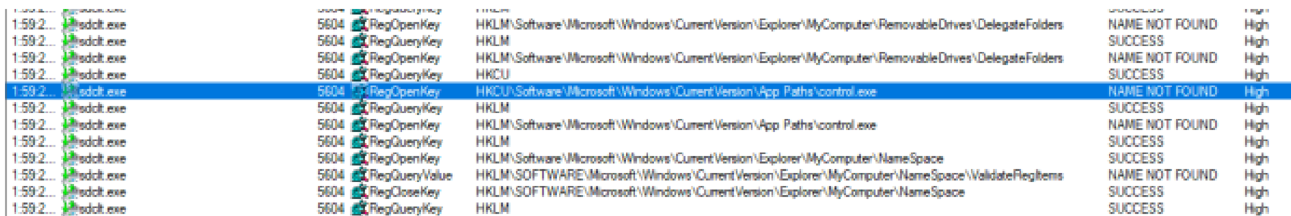
```
<description>manifest file for sdclt</description>
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="amd64"
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
        <requestedPrivileges>
            <requestedExecutionLevel
                level="requireAdministrator"
                uiAccess="false"
            />
        </requestedPrivileges>
    </security>
</trustInfo>
<application  xmlns="urn:schemas-microsoft-com:asm.v3">
    <windowsSettings>
        <dpiAware  xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">true</dpiAware>
        <autoElevate  xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">true</autoElevate>
    </windowsSettings>
</application>
</assembly>
```

**Note: This only works on Windows 10. The manifest for sdclt.exe in Windows 7 has the requestedExecutionLevel set to "AsInvoker", preventing auto-elevation when started from medium integrity.*

When observing the execution flow of sdclt.exe, it becomes apparent that this binary starts control.exe in order to open up a Control Panel item in high-integrity context:

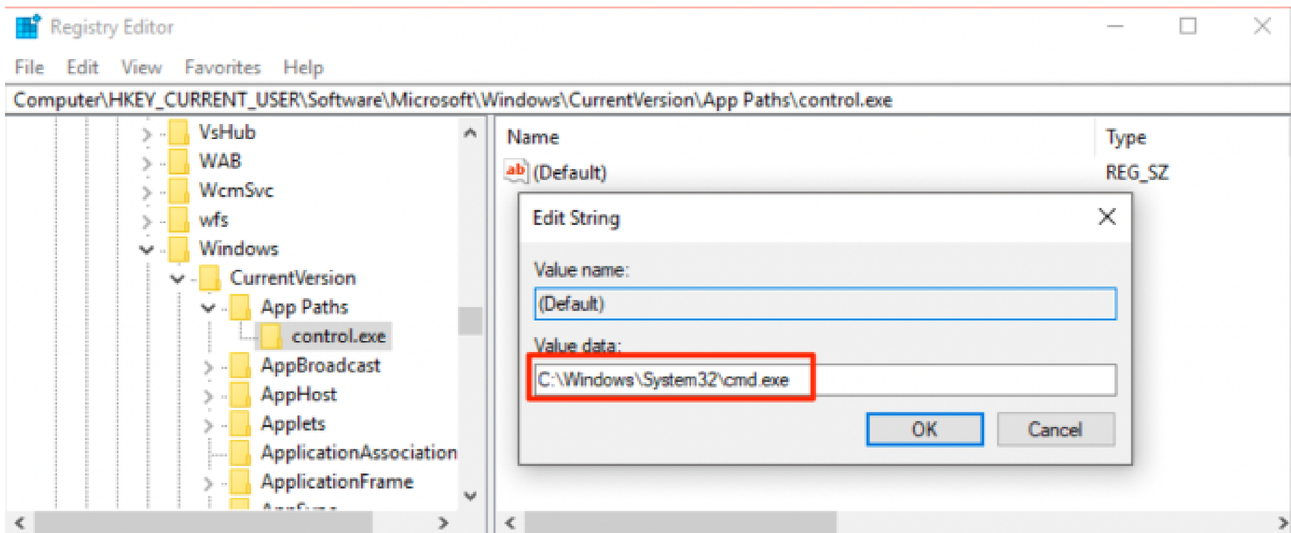


I became curious how sdclt.exe obtains the path to control.exe. Looking again at the execution flow, sdclt.exe queries the App Path key for control.exe within the HKEY_CURRENT_USER hive.

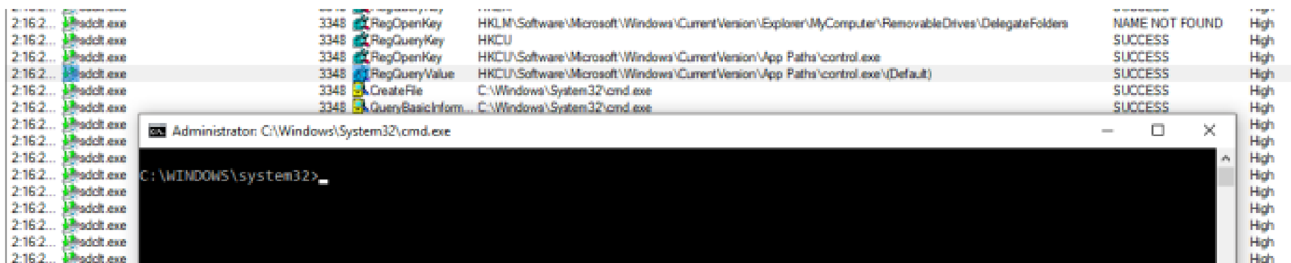


Calls to HKEY_CURRENT_USER (or HKCU) from a high integrity process are particularly interesting. This often means that an elevated process is interacting with a registry location that a medium integrity process can tamper with. In this case, I saw that "sdclt.exe" was querying **HKCU:\Software\Microsoft\Windows\CurrentVersion\App Paths\control.exe.** If you aren't familiar with App Paths in Windows, you can read more about the topic here.

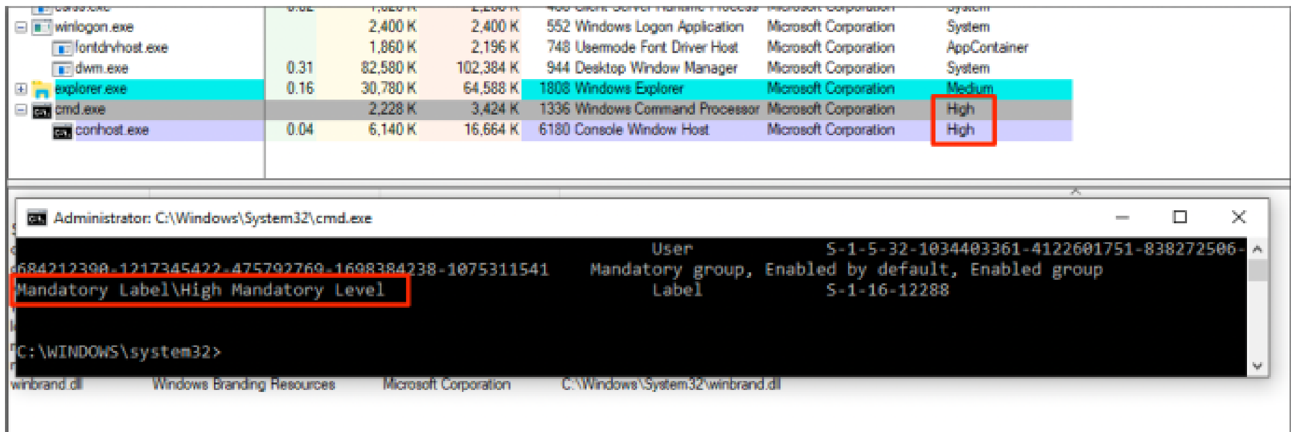
As it stands, sdclt.exe looks for the App Path of control.exe in the HKCU hive. Essentially, this binary is asking "what is the full path of control.exe?". If that key isn't found, it continues the typical Windows search order. Since it is searching in a place that can be modified, we can populate the key.



Guess what happens once sdclt.exe is started again? You guessed it. Sdclt.exe queried our newly created App Paths key for control.exe, resulting in cmd.exe getting returned.



Looking at Process Explorer (or whoami /groups), I was able to confirm that cmd.exe is indeed high integrity:



It is important to note that this technique does not allow for parameters, meaning it requires your payload to be placed on disk someplace. If you try to give the binary any parameters (e.g, C:\Windows\System32\cmd.exe /c calc.exe), it will interpret the entire string as the lpFile value to the [ShellExecuteInfo](#) structure, which is then passed over to [ShellExecuteEx](#). Since that value doesn't exist, it will not execute.

To demonstrate this technique, you can find a script here: https://raw.githubusercontent.com/enigma0x3/Misc-PowerShell-Stuff/master/Invoke-AppPathBypass.ps1

The script takes a full path to your payload. C:\Windows\System32\cmd.exe is a good one to validate. It will automatically add the keys, start sdclt.exe and then cleanup.

This particular technique can be remediated or fixed by setting the UAC level to “Always Notify” or by removing the current user from the Local Administrators group. Further, if you would like to monitor for this attack, you could utilize methods/signatures to look for and alert on new registry entries in **HKCU\Microsoft\Windows\CurrentVersion\App Paths\Control.exe**.

Cheers,
Matt

Source: https://enigma0x3.net/2017/03/14/bypassing-uac-using-app-paths/