Inside Windows 7 User Account Control Mark Russinovich

Standard user accounts provide for better security and lower total cost of ownership in both home and corporate environments. When users run with standard user rights instead of administrative rights, the security configuration of the system, including antivirus and firewall, is protected. This provides users a secure area that can protect their account and the rest of the system. For enterprise deployments, the policies set by desktop IT managers cannot be overridden, and on a shared family computer, different user accounts are protected from changes made by other accounts.

However, Windows has had a long history of users running with administrative rights. As a result, software has often been developed to run in administrative accounts and take dependencies, often unintentionally, on administrative rights. To both enable more software to run with standard user rights and to help developers write applications that run correctly with standard user rights, Windows Vista introduced User Account Control (UAC). UAC is a collection of technologies that include file system and registry virtualization, the Protected Administrator (PA) account, UAC elevation prompts, and Windows Integrity levels that support these goals. I've talked about these in detail in my conference presentations and *TechNet Magazine*UAC internals article.

Windows 7 carries forward UAC's goals with the underlying technologies relatively unchanged. However, it does introduce two new modes that UAC's PA account can operate with and an auto-elevation mechanism for some built-in Windows components. In this post, I'll cover the motivations behind UAC's technologies, revisit the relationship between UAC and security, describe the two new modes, and explain how exactly auto-elevation works. Note that the information in this post reflects the behavior of the Windows 7 release candidate, which is different in several ways from the beta.

UAC Technologies

The most basic element and direct benefit of UAC's technology is simply making Windows more standard-user friendly. The showcase example is the difference between the privilege requirements of setting the time zone on Windows XP and Windows Vista. On Windows XP, changing the time zone—actually, even looking at the time zone with the time/date control panel applet—requires administrative rights.

That's because Windows XP doesn't differentiate between changing the time, which is a security-sensitive system operation, from changing the time zone, which merely affects the way that time is displayed. In Windows Vista (and Windows 7), changing the time zone isn't an administrative operation and the time/date control panel applet separates administrative operations from the standard user operations. This change alone enables many enterprises to configure traveling users with standard user accounts, because users can adjust the time zone to reflect their current location. Windows 7 goes further, making things like refreshing the system's IP address, using Windows Update to install optional updates and drivers, changing the display DPI, and viewing the current firewall settings accessible to standard users.

File system and registry virtualization work behind the scenes to help many applications that inadvertently use administrative rights to run correctly without them. The most common unnecessary uses of administrative rights are the storage of application settings or user data in

areas of the registry or file system that are for use by the system. Some legacy applications store their settings in the system-wide portion of the registry (HKEY_LOCAL_MACHINE\Software) instead of the per-user portion (HKEY_CURRENT_USER\Software), for example, and registry virtualization diverts attempts to write to the system location to one in HKEY_CURRENT_USER (HKCU) while preserving application compatibility.

The PA account was designed to encourage developers to write their applications to require only standard user rights while enabling as many applications that share state between administrative components and standard user components to continue working. By default, the first account on a Windows Vista or Windows 7 system, which was a full administrator account on previous versions of Windows, is a PA account.

Any programs a PA user executes are run with standard-user rights unless the user explicitly elevates the application, which grants the application administrative rights. Elevation prompts are triggered by user activities such as installing applications and changing system settings. These elevation prompts are the most visible UAC technology, manifesting as a switch to a screen with an allow/cancel dialog and grayed snapshot of the desktop as the background.

Accounts created subsequent to the installation are standard user accounts by default that provide the ability to elevate via an "over the shoulder" prompt that asks for credentials of an administrative account that will be used to grant administrative rights. This facility enables a family member sharing a home computer or a more security-conscious user using a standard user account to run applications with administrative rights, provided they know the password to an administrative account, without having to manually switch to a different user logon session. Common examples of such applications include installers and parental control configuration.

When UAC is enabled, all user accounts—including administrative accounts—run with standard user rights. This means that application developers must consider the fact that their software won't have administrative rights by default. This should remind them to design their application to work with standard user rights. If the application or parts of its functionality require administrative rights, it can leverage the elevation mechanism to enable the user to unlock that functionality. Generally, application developers need to make only minor changes to their applications to work well with standard user rights. As the E7 blog post on UAC shows, UAC is successfully changing the way developers write software.

Elevation prompts also provide the benefit that they "notify" the user when software wants to make changes to the system, and it gives the user an opportunity to prevent it. For example, if a software package that the user doesn't trust or want to allow to modify the system asks for administrative rights, they can decline the prompt.

Elevations and Malware Security

The primary goal of UAC is to enable more users to run with standard user rights. However, one of UAC's technologies looks and smells like a security feature: the consent prompt. Many people believed that the fact that software has to ask the user to grant it administrative rights means that they can prevent malware from gaining administrative rights. Besides the visual implication that a prompt is a gateway to administrative rights for just the operation it describes, the switch to a different desktop for the elevation dialog and the use of the Windows Integrity Mechanism, including User Interface Privilege Isolation (UIPI), seem to reinforce that belief.

As we've stated since before the launch of Windows Vista, the primary purpose of elevation is not security, though, it's convenience: if users had to switch accounts to perform administrative operations, either by logging into or Fast User Switching to an administrative account, most users would switch once and not switch back. There would be no progress changing the environment that application developers design for.

So what are the secure desktop and Windows Integrity Mechanism for?

The main reason for the switch to a different desktop for the prompt is that standard user software cannot "spoof" the elevation prompt, for example, by drawing on top of the publisher name on the dialog to fool a user into thinking that Microsoft or another software vendor is generating the prompt instead of them. The alternate desktop is called a "secure desktop," because it's owned by the system rather than the user, just like the desktop upon which the system displays the Windows logon dialog.

The use of another desktop also has an important application compatibility purpose: while built-in accessibility software, like the On Screen Keyboard, works well on a desktop that's running applications owned by different users, there is third-party software that does not. That software won't work properly when an elevation dialog, which is owned by the local system account, is displayed on the desktop owned by a user.

The Windows Integrity Mechanism and UIPI were designed to create a protective barrier around elevated applications. One of its original goals was to prevent software developers from taking shortcuts and leveraging already-elevated applications to accomplish administrative tasks. An application running with standard user rights cannot send synthetic mouse or keyboard inputs into an elevated application to make it do its bidding or inject code into an elevated application to perform administrative operations.

Windows Integrity Mechanism and UIPI were used in Windows Vista for Protected Mode Internet Explorer, which makes it more difficult for malware that infects a running instance of IE to modify user account settings, for example, to configure itself to start every time the user logs on. While it was an early design goal of Windows Vista to use elevations with the secure desktop, Windows Integrity Mechanism, and UIPI to create an impermeable barrier—called a security boundary—between software running with standard user rights and administrative rights, two reasons prevented that goal from being achieved, and it was subsequently dropped: usability and application compatibility.

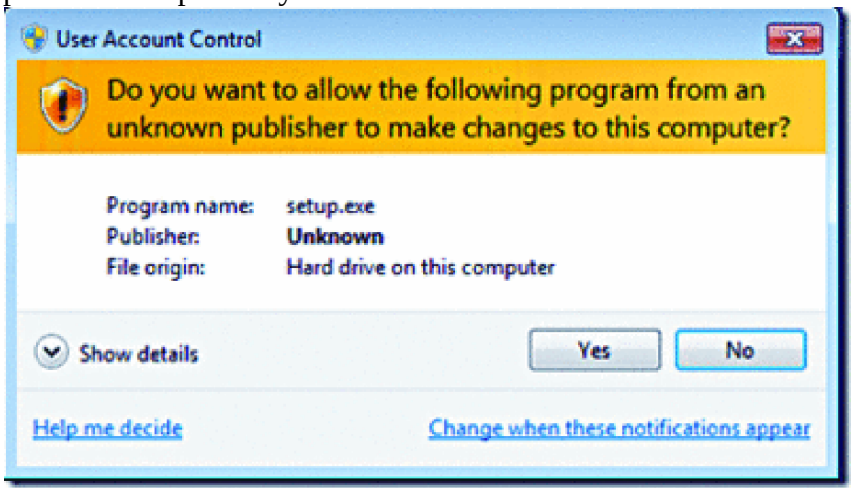

Figure 1 **Showing the executable fi le's name.**

First, consider the elevation dialog itself. It displays the name and publisher of the primary executable that will be granted administrative rights. Unfortunately, while greater numbers of software publishers are digitally signing their code, there are those that aren't, and there are many older applications that aren't signed. For software that isn't signed, the elevation dialog simply shows the executable's file name, which makes it possible for malware already running in a users account and that's watching for an elevation of an unsigned Setup.exe application installer, for example, to replace the executable with a malicious Setup.exe without the user being able to tell (see **Figure 1**).

Second, the dialog doesn't tell the user what DLLs the executable will load once it starts. If the executable resides in a directory under the user's control, malware running with the user's standard rights can replace any associated DLLs in the location that the software will use.

Alternatively, malware could use side-by-side functionality to cause the executable to load malicious versions of application or system DLLs. And unless a user vigilantly clicks the details button and carefully looks at the file path listed for the elevating executable, malware can copy the executable to a similarly named location, for example, \ProgramFiles\Vendor\Application.exe (note the missing space in what should be "Program Files"), where it could control what DLLs the application loads. In **Figure 2**, I've copied a component of Microsoft Network Monitor to the user-created C:\ProgramFiles directory that's controllable by the user and launched it.

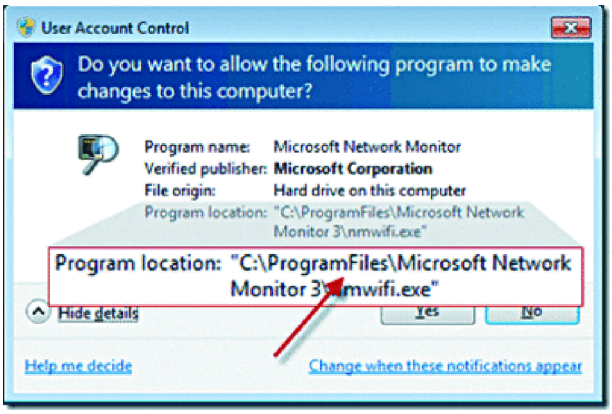

Figure 2 **Launched copy of Microsoft Network Monitor component.**

Finally, for application compatibility, elevated applications share substantial state with the standard user environment that a malicious application could use to influence the behavior of an elevated application. The clearest example of this is the user's registry profile, HKEY_CURRENT_USER (HKCU). That is shared because users expect settings and extensions they register as a standard user to work in elevated applications.

Malware could use shell extensions registered in HKCU to load into elevated applications that use any of the shell browsing dialogs, like File Open and File Save. Other kinds of state are also shared, most notably the Base Named Object namespace, where applications create synchronization and shared memory objects. Malware could take advantage of that sharing to hijack a shared memory object used by an elevated application, for instance, to compromise the application and then the system.

As for the Windows Integrity Mechanism, its effectiveness as a barrier is limited by the elevation issues I've mentioned, but it also has limitations caused by application compatibility. For one,

UIPI doesn't prevent standard user applications from drawing on the desktop, something that could be used to trick the user into interacting with elevated applications in a way that grants malware administrative rights. Windows Integrity Mechanism also doesn't flow across the network.

A standard user application running in a PA account will have access to system resources on a remote system on which that PA account has administrative rights. Addressing these limitations has major application compatibility ramifications. That said, that we are continually looking at ways to improve system security, for instance, improving Protected Mode IE, while at the same time addressing application compatibility issues and working closely with software developers.

So, how much malware protection do you get when you run in a Windows Vista PA account with UAC enabled? First, remember that for any of this to matter, malware has to get onto the system and start executing in the first place. Windows has many defense-in-depth features, including Data Execution Prevention (DEP), Address Space Load Randomization (ASLR), Protected Mode IE, the IE 8 SmartScreen Filter, and Windows Defender that help prevent malware from getting on the system and running.

As for the case where malware somehow does manage to get on a system, because malware authors (like legitimate developers) have assumed users run with administrative rights, most malware will not function correctly. That alone could be considered a security benefit. However, malware that's gotten on a system and that's designed to exploit the opportunities might be able to gain administrative rights the first time the user elevates—but the malware doesn't even need to wait for a "real" elevation because it can precipitate one that would fool even the most security conscious users.

I've demonstrated publicly how malware can hijack the elevation process in my UAC Internals and Windows Security Boundaries presentations (the demo is at minute 1:03 in the security boundaries talk). Remember, though, if malware does start running, it can accomplish most of the things that malware wants to do with just standard user rights, including configuring itself to run every time the user logs on, stealing or deleting all the user's data, or even becoming part of a botnet.

What's Different in Windows 7

I mentioned some of the operations in Windows 7 that can now be performed by standard users, but as the E7 blog post on UAC explains, we also recognized that we could make the Windows experience smoother without sacrificing UAC's goals. Many users complained about the fact that Windows Vista itself frequently asks for administrative rights when they perform common system management operations.

It's in our best interest, because it's in the interest of our customers, to make Windows work well for standard user environments. However, elevation prompts don't educate or encourage us to do so, but they do force users to click a second time through a dialog that the vast majority of users don't read. Windows 7, therefore, set out to minimize those prompts from the default Windows experience and enable users that run as administrators to control their prompting experience.

To do that, we further refactored the system such that someone with standard user rights can execute more tasks, and we reduced the number of prompts in several multi-prompt scenarios (for example, installing an ActiveX control in IE).

Windows 7 also introduces two new UAC operating modes that are selectable in a new UAC configuration dialog (see **Figure 3**). You can open the dialog by going to Control Panel, clicking User Accounts, clicking User Accounts, and then clicking Change User Account Control Settings. (You can also get to it by clicking on the Change When Notifications Appear link on an elevation prompt or by going through the Action Center.)

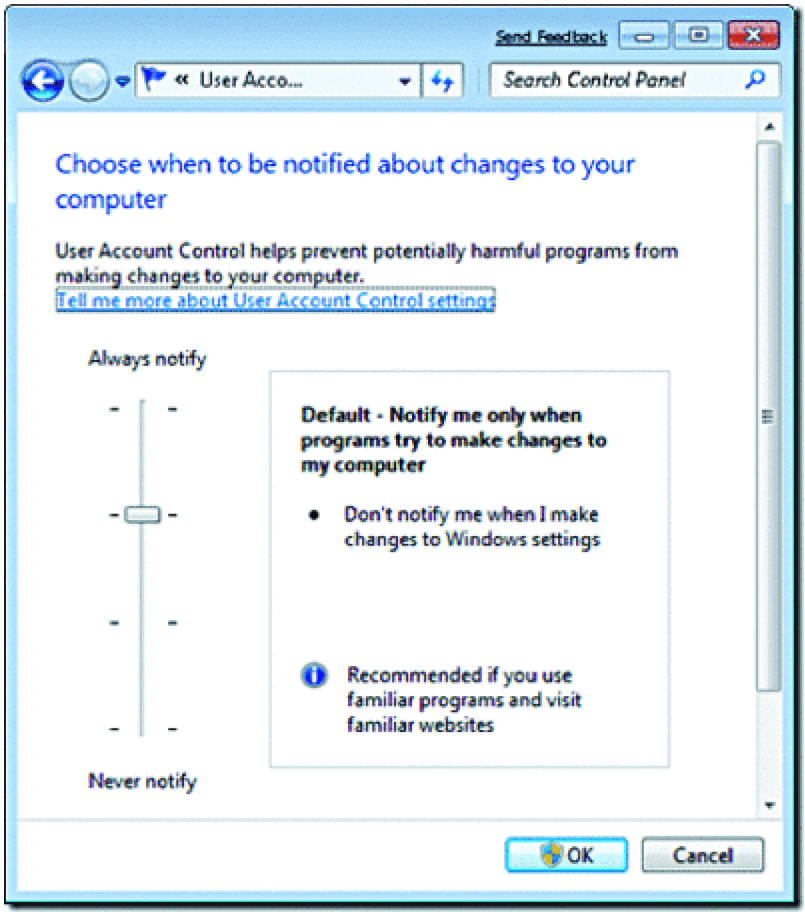

Figure 3 **Two new UAC operating modes that are selectable in a new UAC configuration dialog.**

The default setting, shown in **Figure 3**, is one of the new levels. Unlike Always Notify, which is the selection at the top of the slider and is identical to the default mode in Windows Vista, the Windows 7 default prompts the user only when a non-Windows executable asks for elevation; the behavior for non-Windows elevations is the same as it was for Windows Vista.

The next slider position down is the second new setting and has the same label except with "(do not dim my desktop)" appended to it. The only difference between that and the default mode is that prompts happen on the user's desktop rather than on the secure desktop. The upside of that is that the user can interact with the desktop while a prompt is active, but as I mentioned earlier, the risk is that third-party accessibility software might not work correctly on the prompt dialog. Finally, the bottom slider position turns off UAC technologies altogether, so that all software running in a PA account runs with full administrative rights, file system and registry

virtualization are disabled, and Protected Mode IE is disabled. While there are no prompts at this setting, the loss of Protected Mode IE is a significant disadvantage of this mode.

Auto-Elevation

The reason that elevation of (most) Windows executables in the two middle settings doesn't result in a prompt is that the system "auto elevates" Windows executables. First, what does Windows define as a Windows executable in this context? The answer depends on several factors, but two things must hold: it must be digitally signed by the Windows publisher, which is the certificate used to sign all code included with Windows (it's not sufficient to be signed by Microsoft, so Microsoft software that's not shipped in Windows isn't included); and it must be located in one of a handful of "secure" directories. A secure directory is one that standard users can't modify and they include %SystemRoot%\System32 (e.g., \Windows\System32) and most of its subdirectories, %SystemRoot%\Ehome, as well as a handful of directories under %ProgramFiles% that include Windows Defender and Windows Journal.

Also, depending on whether the executable is a normal .exe, Mmc.exe, or a COM object, auto-elevation has additional rules. Windows executables (as just defined) of the .exe variety auto-elevate if they specify the autoElevate property in their manifest, which is also where applications indicate to UAC that they want administrative rights. Here's the Sysinternals Sigcheck utility dumping the manifest for Task Manager (Taskmgr.exe) with the command "sigcheck –m %systemroot%\system32\taskmgr.exe", which shows that Task Manager is opted in for auto-elevation, as shown in **Figure 4**.

```
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
        <requestedPrivileges>
            <requestedExecutionLevel level="asInvoker"/>
        </requestedPrivileges>
    </security>
</trustInfo>
<asmv3:application>
   <asmv3:windowsSettings xmlns="http://schemas.microsoft.c
ttings">
        <autoElevate>true</autoElevate>
   </asmv3:windowsSettings>
</asmv3:application>
</assembly>
```

Figure 4 **The autoElevate Property**

An easy way to find auto-elevate executables in a directory tree is to use the Sysinternals Strings utility with a command like this:

```
strings –s *.exe | findstr /i autoelevate
```

There is also a hardcoded list of Windows executables that get the auto-elevate treatment. These Windows executables also ship external to Windows 7 and so must be able to run on down-level systems where the presence of the autoexecute property would result in an error. The list includes Migwiz.exe, the migration wizard, Pkgmgr.exe, the package manager, and Spinstall.exe, the Service Pack installer.

The Microsoft Management Console, Mmc.exe, gets special treatment since it hosts many of the system management snap-ins, which are implemented as DLLs. Mmc.exe launches with a command line that specifies a .MSC file that lists the snap-ins MMC is to load. When Windows sees Mmc.exe ask for administrative rights, which it does when launched from a PA account, it validates that Mmc.exe is a Windows executable and then checks the .MSC. To be eligible for auto-elevation, the .MSC file must satisfy the Windows executable criteria (signed by Windows in a secure location) and it must be listed on an internal list of auto-elevate .MSCs. That list includes virtually all the .MSC files shipped with Windows.

Finally, COM objects can specify the need for administrative rights with a registry value in their registry key by creating a subkey named Elevation with a value named Enabled that is set to 1. **Figure 5** shows the registry key for the shell's Copy/Move/Rename/Delete/Link Object that Explorer uses when a user performs a file system operation on a location their account doesn't have permissions for.

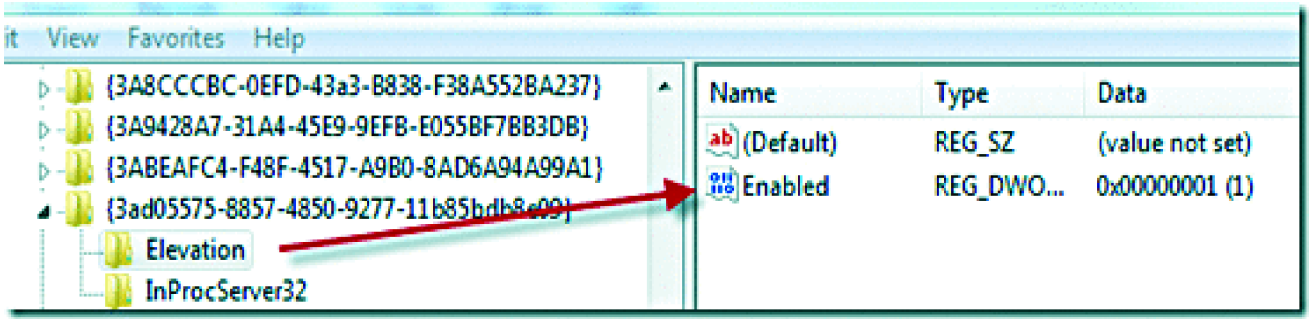

Figure 5 **Shell Registry Key**

For a COM object to be auto-elevated, it must also be a Windows executable and it must have been instantiated by a Windows executable. (The instantiating executable doesn't need to be marked for auto-elevation, though.) When you use Explorer to create a directory in the %ProgramFiles% directory from a PA account, for instance, the operation will auto-elevate because the COM object requests elevation, the object's DLL is a Windows executable, and Explorer is a Windows executable.

Auto-Elevation and UAC's Goals

So what's behind all the special auto-elevation rules? Choosing what to auto-elevate and what not to was guided by the question, "Can an application developer inadvertently or trivially depend on administrative rights by leveraging auto-elevate?" Since Cmd.exe can be used to execute batch scripts via command-line arguments and average users have no need to run command prompts elevated (most don't even know what a command prompt is), it was not manifested for auto-elevation. Similarly, Rundll32.exe, the executable that hosts control panel plug-ins, doesn't auto-elevate in the final release of Windows 7 because its elevation isn't required for any common management tasks, and if it auto-elevated, its ability to host arbitrary DLLs via its command-line could cause a developer to require administrator rights without realizing it.

End users have been asking for Windows to provide a way to add arbitrary applications to the auto-elevate list since the Windows Vista beta. The commonly cited reason is that some third-party application they frequently use forces them to constantly click through an elevation prompt as part of their daily routine. Windows 7, just like Windows Vista, doesn't provide such a

capability. We understand the aggravation, and there might be a legitimate reason that those applications can't run without administrative rights, but the risk is too high that developers will avoid fixing their code to work with standard user rights. Even if the list of what applications get auto-elevated was only accessible by administrators, developers might simply change their application setup program, which requires a one-time elevation, to add their application to the list. We've instead chosen to invest in educating and working closely with application developers to ensure their programs work correctly as a standard user.

Several people have observed that it's possible for third-party software running in a PA account with standard user rights to take advantage of auto-elevation to gain administrative rights. For example, the software can use the WriteProcessMemory API to inject code into Explorer and the CreateRemoteThread API to execute that code, a technique called DLL injection. Since the code is executing in Explorer, which is a Windows executable, it can leverage the COM objects that auto-elevate, like the Copy/Move/Rename/Delete/Link Object, to modify system registry keys or directories and give the software administrative rights. While true, these steps require deliberate intent, aren't trivial, and therefore are not something we believe legitimate developers would opt for versus fixing their software to run with standard user rights. In fact, we recommend against any application developer taking a dependency on the elevation behavior in the system and that application developers test their software running in standard user mode.

The follow-up observation is that malware could gain administrative rights using the same techniques. Again, this is true, but as I pointed out earlier, malware can compromise the system via prompted elevations as well. From the perspective of malware, Windows 7's default mode is no more or less secure than the Always Notify mode ("Vista mode"), and malware that assumes administrative rights will still break when run in Windows 7's default mode.

Conclusion

To summarize, UAC is a set of technologies that has one overall goal: to make it possible for users to run as standard users. The combination of changes to Windows that enable standard users to perform more operations that previously required administrative rights, file and registry virtualization, and prompts all work together to realize this goal. The bottom line is that the default Windows 7 UAC mode makes a PA user’s experience smoother by reducing prompts, allows them to control what legitimate software can modify their system, and still accomplishes UAC’s goals of enabling more software to run without administrative rights and continuing to shift the software ecosystem to write software that works with standard user rights.