THE DANGERS OF PER-USER COM OBJECTS IN WINDOWS

Jon Larimer
IBM X-Force Advanced R&D

Email jlarimer@us.ibm.com

ABSTRACT

Microsoft Windows allows for the registration of COM objects in the user hive of the system registry, and these per-user COM objects generally take precedence over COM objects registered computer-wide. While this offers a lot of flexibility in the way COM objects are installed and allows unprivileged users to install COM components that won't affect other users, there are a few security issues that can be taken advantage of by malicious software.

Because the COM subsystem will load per-user COM objects before the machine-wide objects, malware can register malicious objects in a way that allows for a type of process injection to load malicious code into a target process. In some cases, this is possible even when the process is already running.

This technique can also be used by malware for persistence, without relying on the traditional method of using the 'Run' keys and without requiring administrator privileges. It is also possible to use this technique to create a user-mode rootkit that can hide files and registry keys from other user-mode processes.

And finally, while *Microsoft*'s documentation states that the COM subsystem in *Windows Vista* and later versions will not load per-user COM components for elevated processes, it's possible to bypass this restriction in some cases to execute an elevation of privilege attack.

Luckily, these attacks can be easily detected and there are a variety of ways to mitigate the dangers of per-user COM objects.

PER-USER COM OBJECTS

Per-user Component Object Model (COM) objects are COM components that are registered in the user hive of the system registry (HKEY_CURRENT_USER). Before going into details on how per-user COM objects are used in *Windows*, it's useful to review what COM is and how it works.

The Component Object Model

The Component Object Model, or COM, is an interface used for inter-process communication (IPC) and dynamic object creation [1]. It allows different components to communicate with each other, even components on separate physical computers. Many programming languages offer support for interacting with COM components, and many of the built-in features of *Windows* rely on COM. There are also other *Windows* technologies built on top of COM, such as COM+, DCOM, ActiveX, OLE, and COM Automation. There are some very common *Windows* APIs that are only accessible through COM interfaces, such as Windows Shell Extensions [2].

Registration of COM objects

There are two necessary components for a COM object to work: the registration information in the system registry, and an executable file which is usually a DLL file. When an application wants to instantiate a COM object to interact with it, the COM subsystem uses the system registry to locate the executable code for the object and load it into memory.

Loading and instantiation of COM objects

COM objects are referenced through Class IDs (CLSIDs) that are in the form of Globally Unique Identifiers (GUIDs) [3]. A GUID is a 16-byte value that usually takes the form of a string of hexadecimal digits, for example {01234567-89AB-CDEF-0123-456789ABCDEF}, which is designed to be unique across all computer systems. When an application wants to access a COM object, it makes a call to the CoCreateInstance() function [4], passing a CLSID for the requested object and an Interface ID (IID) for the requested interface. The CLSID is a reference to the actual executable file that contains the code, usually a DLL file, and the IID is a reference to the specific object requested. Once the COM subsystem locates and loads the DLL into memory, it queries the DLL for the requested interface by using the exported DllGetClassObject() function [5].

The COM subsystem locates the executable for a CLSID by examining the system registry, specifically under the key HKEY_CLASSES_ROOT\CLSID. The HKEY_CLASSES_ROOT hive exists for compatibility with 16-bit *Windows* [6], and in modern systems it's actually a 'virtual' key – the hive doesn't physically exist on disk. Instead, it merges information from the machine-wide and per-user COM keys, in HKEY_LOCAL_MACHINE\Software\Classes and HKEY_CURRENT_USER\Software\Classes respectively. When accessing the registry through HKEY_CLASSES_ROOT, all entries from HKEY_CURRENT_USER\Software\Classes are returned, and any entries from HKEY_LOCAL_MACHINE\Software\Classes that aren't duplicates are returned. This means that the current user entries will take precedence over any machine-wide entries when an application or library opens a registry key using the HKEY_CLASSES_ROOT hive, and it follows that per-user COM objects will be loaded before a machine-wide object is loaded if both share the same CLSID.

The only time per-user COM objects are not loaded before a machine-wide object is when the process instantiating the object is running at a privilege level higher than Medium, as is explained below in the section titled 'Protection against privilege elevation attacks in the COM subsystem'.

Abusing per-user COM objects

The danger of per-user COM objects is that any non-administrative process running at a Medium integrity level – that is, any normal user process – can install one. A non-administrator user can make changes to the HKEY_CURRENT_USER registry hive but won't have access to modify the HKEY_LOCAL_MACHINE hive.

Malware can take advantage of the fact that per-user COM objects are loaded before machine-wide COM objects to perform a few different malicious attacks against a system. The basic attack is that a malware author can examine which COM objects are loaded by a process, such as *Windows Explorer* or a web browser, and then install a per-user COM

object with the same CLSID. When the COM subsystem, on behalf of the target process, attempts to locate the executable file that contains that CLSID's object, it will find and load the per-user object first.

PROCESS INJECTION

Process injection is a technique commonly used by malware to run malicious code in a separate, non-malicious process running on a system. A typical example would be malware that uses the WriteProcessMemory() and CreateRemoteThread() API calls to cause a malicious DLL file to be loaded into the *Windows Explorer* process [7]. There are several reasons that malware would want to inject code into another process. The first is to hide from the user. An astute computer user might notice an unfamiliar process running on their system, so many modern samples of malware will inject their code into another process. Another reason is to bypass personal firewall software that is able to block network communication from unknown processes. A piece of malware can inject their code into a process which allows network communication – such as a web browser – to communicate with a command and control server or to exfiltrate data. The Zeus and SpyEye malware families are known to do this [8].

COM objects offer another method to get malicious code running inside another process. If a piece of malicious software can cause a running process to instantiate a COM object, a malicious DLL can be loaded into the process's address space and executed. The number of target processes that allow this is limited – many applications will instantiate required COM objects when starting up but not after the process has been running for a while. One notable process that can be an attractive target for COM object process injection is the Windows Shell, explorer.exe.

Microsoft allows the functionality of the Windows Shell to be extended with an API known as Windows Shell Extensions. These extensions are COM objects that can be registered in either the machine-wide or per-user hive of the registry. The COM objects must implement a specific interface to be used as an extension. A common example is icon handlers. An icon handler can be registered that allows a COM component to display a custom icon for a specific file type when browsing folders in *Windows Explorer*. When one of those file types is shown in a folder, *Explorer* will instantiate the COM object to obtain an icon for that file [9]. Vulnerabilities in shell extension handlers have been exploited before. The Stuxnet worm used a vulnerability in the icon handler for LNK files to execute code and infect systems [10].

This ability to load shell extensions on demand can also be leveraged to inject an arbitrary DLL file into the *Explorer* process. Once a shell extension handler is registered in the registry, an application can use the SHChangeNotify() API call to cause *Windows Explorer* to search for new handlers [11]. If the handler is for an object on the desktop, the object will be loaded immediately. Besides registering icon handlers for file extensions, it's possible to register icon handlers for other objects such as the Recycle Bin. This can be a very effective shell extension handler to register since most desktops will have a Recycle Bin icon on them.

This method of injecting code into *Explorer* isn't a huge security risk itself – malware can just as easily use the well known WriteProcessMemory() and CreateRemoteThread() technique since explorer.exe runs at the same privilege level as most other user processes.

While many existing malware families take advantage of process injection techniques to steal user data from within processes, we aren't aware of any specific malware families that use per-user COM objects to inject the malicious payload.

MALWARE PERSISTENCE

When a piece of malware is executed on a system, one of the first things that happens is that the malware will try to achieve persistence on the system so it will always start when the computer boots up or a user logs on. In the early days of computer viruses, the boot sector on floppy disks was modified. In the era of *Windows*, the Run keys in the registry were used. There are now a wide variety of techniques that malware will use to achieve persistence on a machine – installing drivers or services, using a bootkit on the MBR, making use of the Startup folder, the registry's Run keys, or a number of other possible registry keys that can trigger loading of malicious code.

Per-user COM objects are another possible way that malware can achieve persistence with limited permissions on the system. *Windows Explorer* will always load certain COM objects when it starts up, so malicious software could install a per-user COM object to hijack the loading of a machine-wide COM object.

If the goal of malware is to monitor web traffic, it could set up per-user COM objects that are loaded by web browsers. This makes it possible for malware to be loaded into memory while the user is browsing the web instead of automatically with the computer booting or immediately after logon. This can allow malware to be stealthier – only running when necessary.

This method of malware persistence can also be used to implement a user-mode rootkit that hides files and registry keys from system administration tools. If an attacker can cause tools like Registry Editor or Process Explorer to load a malicious per-user COM object into their process space, the malicious code could hook functions that display data in the user interface, hiding the existence of the malware. Although this technique can't be used to hide malware from kernel-mode drivers that monitor the system at a lower level, it can be used to hide data from the user-mode UI that communicates with the driver. For example, the Process Monitor filter drivers will see accesses to a certain registry key, but a user-mode rootkit could prevent the user interface from displaying those entries.

At this time, we aren't aware of any specific malware families that take advantage of per-user COM objects for malware persistence.

ELEVATION OF PRIVILEGE

While the COM subsystem will prevent elevated processes from loading per-user COM objects in *Windows Vista* and *Windows 7*, it's still possible to trick an elevated process into loading them in some cases. This allows a privilege escalation attack, where a Medium integrity level process is able to get arbitrary code to execute at a High integrity level. This type of attack only works when the user already has administrative access to a machine, for example by being a member of the

Administrators group, but can still be dangerous because it bypasses certain security controls in *Windows*.

Windows integrity levels and UAC

Windows Vista saw the introduction of integrity levels and User Account Control (UAC) into the operating system. The *Windows* integrity mechanism restricts the ability of untrusted applications to access or modify the system without user consent [12]. This allows different applications running under the same user account to have different levels of permission to access the system. For example, a normal user application running under an Administrator account will have a *Medium* integrity level that allows it to write to files in the user's home directory and the HKEY_CURRENT_USER hive of the registry, but it doesn't have permission to write to the root directory of the system volume. A process running as the same user, but with a *High* integrity level, does have permission to write to that location. Before integrity levels were introduced, an Administrator user had unrestricted access to the system.

UAC provides a way for applications to run at an elevated privilege level. The way it works is that Administrative users are granted two separate tokens during logon – a regular user token with a medium integrity level, and an administrative token with a high integrity level and additional administrative privileges. Most processes run by the administrative user will run at the medium integrity level, but there are a number of ways that they can run an elevated process using the high integrity level administrative token. Using the Run as Administrator right-click option will launch a process with the administrative token and an application can specify in its manifest file that it requires running at an elevated privilege level. In both of those cases, a UAC prompt is normally displayed to the user asking for confirmation to run the process with administrative privileges.

In *Windows 7*, *Microsoft* introduced a change to UAC that allows some programs to 'auto elevate'. This means that an administrative user can run one of these programs at a high integrity level without seeing the UAC prompt by default. This is done by including the undocumented AutoElevate element in the application's embedded manifest. An application needs to be digitally signed by *Microsoft* for AutoElevate to work, so it's not possible for third party developers or malware authors to use this feature to automatically elevate their own applications.

The level of UAC prompting is configurable. The current default is 'Notify me only when programs try to make changes to my computer; don't notify me when I make changes to Windows settings.' This means that non-*Microsoft* applications that request elevation will result in a UAC prompt, but bundled *Windows* applications can run elevated with no prompting. At the highest security level, the AutoElevate mechanism is disabled and users are still prompted for elevation when an AutoElevate process is launched.

Previous work

Before getting into privilege elevation issues associated with per-user COM objects, it may be useful to review similar research. In 2009, before *Windows 7* was officially released, Leo Davidson published research showing that the AutoElevate functionality could be used to elevate malicious code [13]. Davidson's proof-of-concept code [14] makes use of a couple of different security vulnerabilities to launch a Command Prompt process with Administrative privileges in *Windows 7* without asking the user for elevation through a UAC prompt. The first part of the exploit takes advantage of the ability for certain COM objects to automatically elevate when running from *Microsoft*-signed applications. The example code uses the IFileOperation object, launched from code injected into the explorer.exe process. The second part of the exploit makes use of a UAC whitelisted application – sysprep.exe – to load a DLL dropped by the first part of the exploit. sysprep.exe exists in a subdirectory of the Windows System directory, in \System32\sysprep, and loads a certain DLL file when it runs. That DLL normally exists in the System32 directory, but sysprep.exe will first look in its own directory for the DLL, which is the standard DLL search order in *Windows*. Davidson's exploit uses IFileOperation to drop a fake copy of this DLL into the sysprep directory, which then gets loaded by sysprep.exe. The fake DLL file launches the Command Prompt application. Because sysprep.exe runs with elevated privileges, the Command Prompt is also elevated. Davidson's proof-of-concept attack still works as of May 2011, almost two years after it was published, but we haven't seen any malware that takes advantage of that specific attack vector to increase privileges on a system.

Protection against privilege elevation attacks in the COM subsystem

Along with the introduction of integrity levels and UAC in *Windows*, *Microsoft* changed how the COM subsystem locates COM objects in processes running at an elevated integrity level. The documentation at *MSDN* [15] states:

> 'Beginning with Windows Vista® and Windows Server® 2008, if the integrity level of a process is higher than Medium, the COM runtime ignores per-user COM configuration and accesses only per-machine COM configuration. This action reduces the surface area for elevation of privilege attacks, preventing a process with standard user privileges from configuring a COM object with arbitrary code and having this code called from an elevated process.'

Investigation shows that this is generally correct – a process running at elevated privilege will not attempt to load a COM object that was registered in the user hive of the registry when using the CoCreateInstance() API call to instantiate the object. When running in an elevated process, it will use the HKEY_LOCAL_MACHINE hive directly when looking for an object instead of using HKEY_CLASSES_ROOT.

Bypassing the protection

During our research, we discovered that the protection offered by the COM subsystem to prevent per-user COM objects from being loaded by high integrity processes is not entirely sufficient. There is code in the Windows Shell API which is designed to load COM objects used by the shell. Specifically, a function named SHCoCreateInstance() is exported that *Explorer* and other processes can use to load shell-specific COM objects that are implemented within shell32.dll [16], but can also be used to load COM objects external to that library. SHCoCreateInstance() can bypass some of the

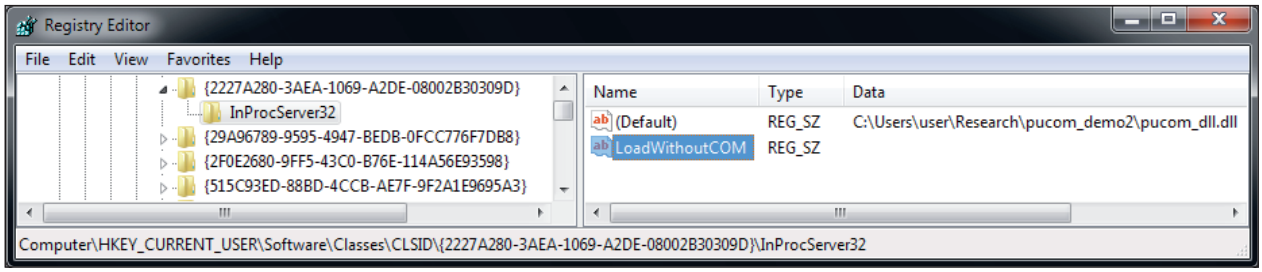

Figure 1: LoadWithoutCOM value.

protections offered by the COM subsystem. When looking for the executable file that contains a COM object's implementation, it will check for the existence of a registry value named LoadWithoutCOM in the InProcServer32 subkey. Figure 1 shows what the existence of this value looks like in RegEdit. If this value exists, code in shell32.dll will call LoadLibraryEx() on the specified DLL file. The reason that this is bad is because the registry is accessed using the HKEY_CLASSES_ROOT hive, which as mentioned above shows a merged view of the machine-wide and per-user class keys in the registry. This makes it possible for a process running at a High integrity level to load and execute a DLL file for a per-user COM object, bypassing the COM subsystem's protection.

We developed two different proof-of-concept programs to demonstrate this vulnerability. The first program demonstrates a UAC Hijacking attack that requires a user to launch a non-malicious program and approve a UAC prompt in order to execute a payload at a High integrity level. The non-malicious program could be a software installer or an application bundled with *Windows*, such as Registry Editor, that prompts for elevation. The second concept program demonstrates a full UAC Bypass attack that launches a malicious process at a High integrity level without a UAC prompt.

The UAC Hijacking proof-of-concept works by registering a COM object used by a Shell API call made by the *Windows* Registry Editor (RegEdit, or regedit.exe). When RegEdit starts up, a UAC prompt is presented to the user asking for elevation. If the user accepts, RegEdit runs at a High integrity level. During startup, RegEdit makes a call to SHGetStockIconInfo() [17]. This function eventually makes a call to SHCoCreateInstance() looking for a COM object with a certain CLSID. The proof-of-concept exploit registers a per-user COM object for that CLSID that includes the LoadWithoutCOM value. When a user attempts to run RegEdit, the 'malicious' DLL gets loaded into RegEdit's process and launches *Notepad* running at a High integrity level. As mentioned before, this isn't specific to RegEdit and an attacker could also take advantage of a shell COM object that's instantiated when certain software installers run.

The UAC Bypass proof-of-concept takes advantage of a *Windows* executable with the AutoElevate ability, similar to the attack described by Davidson. Through a lot of trial and error, it was found that running the command 'printui.exe' with a specific command line argument would launch the process with a High integrity level, where it would then make an API call that instantiates a COM object with SHCoCreateInstance(). The demonstration exploit registers a per-user COM object with a CLSID that causes a DLL to get loaded by printui.exe at a High integrity level without a UAC prompt. An attacker using this kind of attack wouldn't have to wait for a user to run a process that requires manual elevation through the UAC prompt – the elevation happens immediately and without prompting. Figure 2 shows what Process Monitor shows while printui.exe is loading this COM object. Take note of the High integrity level, the check for the LoadWithoutCOM value, and the Load Image operation on pucom_dll.dll.

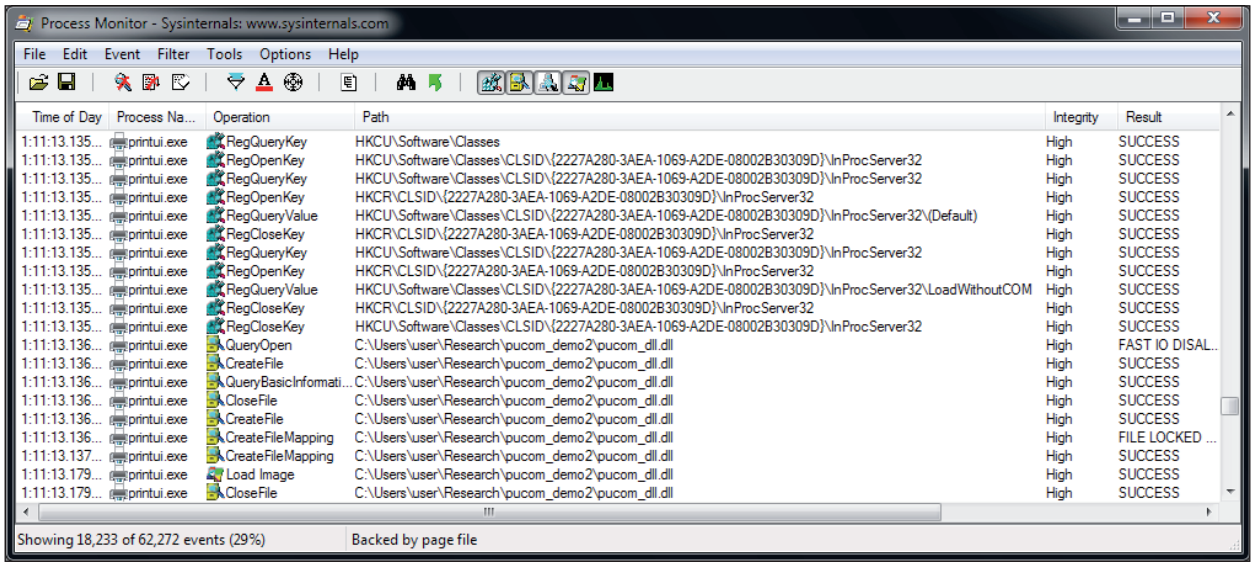

Figure 2: printui.exe loading a 'malicious' DLL.

The LoadWithoutCOM per-user COM object elevation of privilege issue was reported to the *Microsoft Security Response Center* (*MSRC*) in March of 2011. They acknowledged that this is a privilege escalation vulnerability, but declined to issue a bulletin or produce a patch since the exploit requires a user that already has administrative privileges. This wasn't a surprise considering that the Davidson's UAC bypass vulnerability from 2009 hasn't been fixed either.

MITIGATIONS

There are a variety of techniques that can be used by system administrators and anti-malware software vendors to detect or even prevent the exploitation of per-user COM objects for malicious purposes.

Application whitelisting

One of the most effective ways to prevent unauthorized and potentially malicious code from running on a PC is through application whitelisting. *Windows 7* provides the AppLocker feature that not only has the ability to prevent the execution of processes, but also the loading of DLLs by allowed processes. This means it can be used to restrict which shell extensions are able to be loaded by *Explorer*. DLL blocking is not enabled by default however, and can take some time and effort to set up correctly. Information on configuring AppLocker can be found in the MSDN Library [18].

Registry scanning

A tool could be created to scan the system registry to look for registered per-user COM objects. If one is found that also has a machine-wide CLSID registered with a different DLL, there is a definite cause for suspicion. A script that implements this technique is in the Appendix. However, this might not catch all possible attempts at exploiting per-user COM objects. For example, a malicious application could register a *Windows* Shell Extension that doesn't have a corresponding machine-wide COM object. In fact, it's perfectly legitimate for a Shell Extension to register itself in the per-user hive of the registry. In that case traditional anti-virus scanning techniques should be used to detect if a DLL file is malicious or not before it executes in the context of *Explorer*.

To detect malware attempting to take advantage of the privilege elevation vulnerability mentioned above, one can scan for the existence of the LoadWithoutCOM registry value. There's really no reason that value should exist on a modern *Windows* system, but it's still entirely possible that some legacy application may be using it.

Registry monitoring

A tool that monitors registry reads and writes could watch for access to the per-user registry hive from elevated processes and automatically block access to them. Of course, this may adversely impact some software so more research into this mitigation technique needs to be done. Registry monitoring can be performed using Registry Filtering Drivers as documented in the MSDN Library [19].

Preventing privilege elevation exploits

The privilege elevation attacks outlined in this paper are only exploitable when a user account already has administrative access – they're effectively just a way around the UAC feature in *Windows*. While it's always a good idea to use non-administrative accounts when working online, it's not always practical, especially for home users.

To prevent the UAC Bypass attack, users and administrators can set the UAC notification setting to 'Always Notify'. With that setting enabled, AutoElevate is effectively disabled, so users will need to approve all elevations. This also mitigates Davidson's exploit, as long as the user can recognize that something malicious is happening and they choose 'No' when prompted with the UAC dialog.

Preventing a UAC Hijack attack can be accomplished by scanning the registry for per-user COM objects for CLSIDs that also exist as machine-wide COM objects before launching a program that requires elevation. To stop the attack from being successful, either the user could be warned that something malicious could happen or the executable could be blocked from executing. Another possibility is automatically removing the registry entry for the per-user COM object.

CONCLUSION

Research has shown that allowing the COM subsystem to give precedence to per-user COM objects is a security risk for desktop *Windows* users. Although some of these threats can be mitigated with administrative policies in enterprise environments, home users are still at risk. We hope that the information presented in this paper will help security software vendors detect and prevent malicious software abusing the techniques that were outlined here.

REFERENCES

[1] Microsoft. COM: Component Object Model. Microsoft.com. http://www.microsoft.com/com/default.mspx.

[2] Creating Shell Extension Handlers. MSDN Library. [Cited: May 6, 2011.] http://msdn.microsoft.com/en-us/library/cc144067(v=vs.85).aspx.

[3] COM Technical Overview. MSDN Library. http://msdn.microsoft.com/en-us/library/ff637359(v=VS.85).aspx.

[4] CoCreateInstance Function. MSDN Library. http://msdn.microsoft.com/en-us/library/ms686615(v=vs.85).aspx.

[5] DllGetClassObject Entry Point. MSDN Library. http://msdn.microsoft.com/en-us/library/ms680760(v=vs.85).aspx.

[6] HKEY_CLASSES_ROOT Key. MSDN Library. [Cited: May 6, 2011.] http://msdn.microsoft.com/en-us/library/ms724475(v=vs.85).aspx.

[7] Kuster, R. Three Ways to Inject Your Code into Another Process. CodeProject. 20 August 2003. http://www.codeproject.com/KB/threads/winspy.aspx.

[8] Nayyar, H. Clash of the Titans: ZeuS v SpyEye. SANS Institute InfoSec Reading Room. http://www.sans.org/reading_room/whitepapers/malicious/clash-titans-zeus-spyeye_33393.

[9] Microsoft. How to Create Icon Handlers. MSDN Library. http://msdn.microsoft.com/en-us/library/cc144122(v=VS.85).aspx.

[10] Saade, T. The Stuxnet Sting. Microsoft Malware Protection Center. http://blogs.technet.com/b/mmpc/archive/2010/07/16/the-stuxnet-sting.aspx.

[11] Microsoft. SHChangeNotify Function. MSDN Library. http://msdn.microsoft.com/en-us/library/bb762118(v=vs.85).aspx.

[12] What is the Windows Integrity Mechanism? MSDN Library. http://msdn.microsoft.com/en-us/library/bb625957.aspx.

[13] Davidson, L. Windows 7 UAC whitelist: Code-injection Issue (and more). Pretentious Name. [Cited: May 5, 2011.] http://www.pretentiousname.com/misc/win7_uac_whitelist2.html.

[14] Windows 7 UAC whitelist: Detailed Description. Pretentious Name. [Cited: May 5, 2011.] http://www.pretentiousname.com/misc/W7E_Source/win7_uac_poc_details.html.

[15] Microsoft. Application Compatibility: UAC: COM Per-User Configuration. MSDN Library. [Cited: April 14, 2011.] http://msdn.microsoft.com/en-us/library/bb756926.aspx.

[16] SHCoCreateInstance Function. MSDN Library. [Cited: May 5, 2011.] http://msdn.microsoft.com/en-us/library/bb762124(v=vs.85).aspx.

[17] SHGetStockIconInfo Function. MSDN Library. [Online] http://msdn.microsoft.com/en-us/library/bb762205(v=vs.85).aspx.

[18] AppLocker. MSDN Library. http://technet.microsoft.com/en-us/library/dd723678(WS.10).aspx.

[19] Filtering Registry Calls. MSDN Library. [Cited: May 6, 2011.] http://msdn.microsoft.com/en-us/library/ff545879(v=VS.85).aspx.

APPENDIX

VBScript code to scan for per-user COM objects that overlap machine-wide objects

```
' puscan.vbs
'
' Script to locate per-user COM objects that will be loaded
' instead of existing machine-wide COM objects
'
' Run with: cscript puscan.vbs
'
' Jon Larimer <jlarimer@us.ibm.com>

const HKEY_CURRENT_USER = &H80000001
const HKEY_LOCAL_MACHINE = &H80000002

clsidpath = "Software\Classes\CLSID"
set reg=GetObject("winmgmts:{impersonationLevel=impersonate}!\\.\root\default:StdRegProv")

' enumerate per-user COM CLSIDs
reg.EnumKey HKEY_CURRENT_USER, clsidpath, clsids

if IsNull(clsidss) then
   WScript.Echo "No per-user COM objects found"
   WScript.Quit(0)
end if

count = 0

for each clsid in clsids
       dllpath = clsidpath + "\" + clsid + "\InProcServer32"

       ' look for the path to the DLL that contains the code
   reg.GetStringValue HKEY_CURRENT_USER, dllpath, Null, pudll
       if not IsNull(pudll) then
               ' now look for a machine-wide COM object with the same CLSID
               reg.GetStringValue HKEY_LOCAL_MACHINE, dllpath, Null, mwdll
               if not IsNull(mwdll) and (not mwdll = pudll) then
                       WScript.Echo "Possible per-user COM object hijack:"
                       WScript.Echo "  CLSID: " + clsid
                       WScript.Echo "  Per-user DLL: " + pudll
                       WScript.Echo "  Machine-wide DLL: " + mwdll

                       count = count + 1
               end if
       end if
next

if count = 0 then
       WScript.Echo "No per-user COM objects found that overlap machine-wide objects"
       WScript.Quit(0)
end if
```