Vista and ActiveX control

Prepared By: **Su Yong Kim**

Dong Hyun Lee

Do Hoon Lee

Abstract

This article covers security issues of ActiveX control on Windows Vista. Windows Vista has some new security mechanisms such as the UAC (User Account Control) and Protected Mode. Therefore, many old exploit codes for ActiveX control do not work on Windows Vista. However, after close investigation, we recognized that only a few things had changed. In addition, some developers are writing their ActiveX control for Windows Vista in unsecure ways. This makes Windows Vista security mechanisms useless. In this article, we will describe what changes have been made to ActiveX control on Windows Vista.

Introduction

ActiveX control is executed in Internet Explorer. Therefore, ActiveX control has the same privileges as Internet Explorer. On Windows XP, Internet Explorer is operated with the privilege of the user who executes it. Because most XP users login with the account in administrator group, Internet Explorer has administrator privileges. Therefore, ActiveX control can do everything on Windows XP. It can write and read any files or registry keys. It can execute any processes with administrator privilege. Therefore, if ActiveX control is successfully exploited on Windows XP, malicious users can obtain administrator privilege of the victim's system.

On Windows Vista, Internet Explorer is run at low integrity under Protected Mode. ActiveX control with low integrity can only read most files or registry keys, but cannot write many sensitive data on the user's machine. Table 1 compares things ActiveX control

can do between Windows XP and Windows Vista.

Table 1. What ActiveX control can and cannot do

Operating System / Activity	XP	Vista
Writing a file/registry key with low integrity	N/A	Possible
Writing a file/registry key with medium integrity and above	**Possible**	**Impossible**
Executing a process with low integrity	N/A	possible
Executing a process with medium integrity and above	**Possible**	**User agreement required**
Reading a file/registry key	Possible	Possible

Differences of Vulnerabilities on Windows Vista

Except file/registry reading vulnerability, there are some vulnerability differences on Windows Vista.

First, it is more difficult to install a malicious program exploiting file/registry writing vulnerabilities on Windows Vista. File writing vulnerability can be misused to create a malicious program under the Startup folder on Windows XP. However, file writing vulnerability cannot be done on Windows Vista because ActiveX control doesn't have write permission on the Startup folder. Similarly, Registry writing vulnerability cannot be misused to create a registry key with a malicious command executed at boot time on Windows Vista.

Second, the process execution vulnerability caused by CreateProcess can be successfully exploited only with user agreement on Windows Vista. Windows Vista requires a user to agree to privilege elevation because CreateProcess usually runs a process with medium integrity and above.

Third, Shellcodes for Buffer overflow vulnerability on Windows XP need to be modified to work on Windows Vista because they have no write permission on most resources.

1. File/Registry Writing Vulnerability on Windows Vista

ActiveX control cannot write any files in a sensitive folder such as the Startup folder because it has medium integrity. It is not allowed to create a malicious file in the Startup

folder by exploiting file writing vulnerability on Windows Vista. However, developers suffer the same problems as malicious users. If ActiveX control is installed in the "Program Files" folder, it cannot be updated without user agreement. Therefore, some developers install ActiveX control in a low integrity folder to update it silently.

If ActiveX control is installed in a low integrity folder, malicious users can run a program misusing file writing vulnerability. Malicious users can overwrite a sensitive file such as a DLL file or a configuration file in a low integrity folder. Whenever the overwritten DLL file is loaded, user-privileged malicious programs can be run at low integrity.

2. Process Execution Vulnerability on Windows Vista

Whenever privilege elevation occurs, Windows Vista requires user agreement. On Windows Vista, CreateProcess runs processes at medium integrity and above, but CreateProcessAsUser can run at low integrity. Therefore, process execution vulnerability caused by CreateProcess can be successfully exploited only with user's agreement. However, process execution vulnerability caused by CreateProcessAsUser can be misused without user's agreement if CreateProcessAsUser is used in ActiveX control to run a process at low integrity.

In conclusion, user-privileged malicious programs can be run at low integrity by exploiting process execution vulnerability on Windows Vista.

One interesting fact is that Windows Vista requires user agreement to execute an unsigned program. Malicious users usually execute mshta.exe with a remote hta file to install a malicious program. While interpreting an hta file, a malicious program is created and executed in the victim's system. Figure 1 shows a VBScript code to execute a malicious program.

```
Set shell = CreateObject("WScript.Shell")
Shell.Run "sweetlie.exe"
```

Figure 1. VBScript code to execute malicious program

As mentioned above, if sweetlie.exe doesn't have a valid signature, Windows Vista requires user agreement to run sweetlie.exe. Figure 2 shows the security warning window.

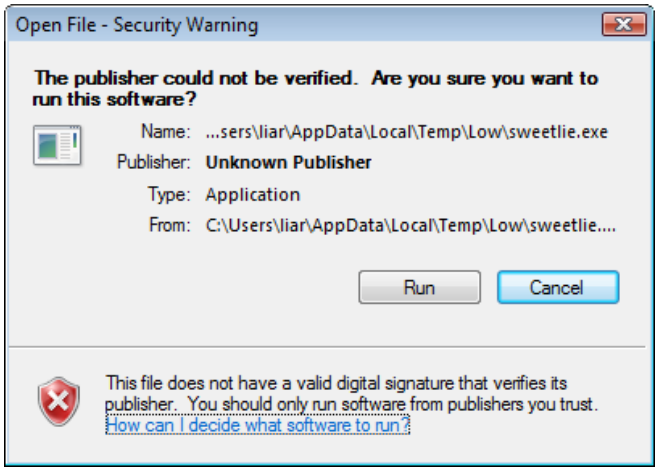

Figure 2. Security warning window

However, this security protection can easily be evaded. Malicious users can use a signed program such as cmd.exe to execute a malicious program. Figure 3 shows an example that uses cmd.exe to execute a malicious program.

```
Set shell = CreateObject("WScript.Shell")
Shell.Run "cmd.exe /c sweetlie.exe"
```

Figure 3. VBScript code using cmd.exe

3. Buffer Overflow Vulnerability on Vista

On Windows Vista, buffer overflow vulnerabilities are nearly the same as on Windows XP. Buffer overflow vulnerability of ActiveX control is exploited via heap spraying method. Heap spraying method doesn't depend on the address of loaded DLL files. Therefore, Address space layout randomization enabled on Windows Vista doesn't prevent heap spraying method from working.

However, many old exploit codes for buffer overflow vulnerabilities don't work on Windows Vista. Because Internet Explorer has low integrity, two rules should be followed. If a new process needs to be executed, CreateProcessAsUserA should be used, not CreateProcessA. If a new file needs to be created, it should be done in a folder with low integrity.

Figure 4 shows a typical shellcode procedure for Windwos XP.

1. Find the address of kernel32.dll
2. Find the addresses of some API functions in kernel32.dll
   - LoadLibraryA, CreateFileA, WriteFile, CloseHandle, CreateProcessA, ExitProcess
3. Call LoadLibraryA for wininet.dll
4. Find the addresses of some API functions in wininet.dll
   - InternetOpenA, InternetOpenUrlA, InternetReadFile
5. Call InternetOpenA & InternetOpenUrlA
6. Call CreateFileA & InternetReadFile &WriteFile & CloseHandle
7. Call CreateProcessA & ExitProcess

Figure 4. Shellcode procedure for Windows XP

In figure 4, shellcode downloads a malicious file from a remote web server, and execute it. However, On Windows Vista, CreateFileA will fail because a downloaded file cannot be created in a folder with medium integrity. Similarly, CreateProcessA requires user agreement to execute process.

In figure 5, shellcode was modified to work on Windows Vista

1. Find the address of kernel32.dll
2. Find the addresses of some API functions in kernel32.dll
   - LoadLibraryA, CreateFileA, WriteFile, CloseHandle, ExitProcess, GetTempPathA
3. Call LoadLibraryA for wininet.dll
4. Find the addresses of some API functions in wininet.dll
   - InternetOpenA, InternetOpenUrlA, InternetReadFile
5. Call InternetOpenA & InternetOpenUrlA
6. Call GetTempPathA
7. Call CreateFileA & InternetReadFile & WriteFile & CloseHandle
8. Call LoadLibraryA for advapi32.dll
9. Find the addresses of CreateProcessAsUserA in advapi32.dll
10. Call CreateProcessAsUserA & ExitProcess

Figure 5. Shellcode procedure for Windows Vista

In figure 5, GetTempPathA returns %Temp%₩Low because Internet Explorer's environment variable is modified under protected mode. A malicious file is created in %Temp%₩Low folder with low integrity. CreateProcessAsUserA runs it at low integrity.

In figure 6, we see that CreateProcessAsUserA has one more argument, hToken, than CreateProcessA. If hToken is set to NULL, CreateProcessAsUserA runs a process at the

same integrity as Internet Explorer.

```
BOOL WINAPI CreateProcessAsUserA(
  HANDLE hToken,
  LPCSTR lpApplicationName,
  LPSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Figure 6. Prototype of CreateProcessAsUserA

How to restart a malicious program

If process execution vulnerability or buffer overflow vulnerability is successfully exploited, malicious user can execute a user-privileged malicious program at low integrity. This program can steal most files and registry information. However, it cannot be executed again at boot time, because medium integrity is required to register any executable file as a startup program.

Like file/registry writing vulnerability, a malicious user can overwrite DLL files with low integrity to restart malicious programs whenever they are loaded by another process. If an overwritten DLL file is loaded by a higher-privileged process, privilege of malicious program is elevated and then malicious program can be registered as a startup program.

We developed a tool to monitor a process loading DLL files with low integrity. It hooks LoadLibraryA and LoadLibraryW API. If a loaded DLL file is low integrity, it shows integrity level of the process and path of the DLL file.

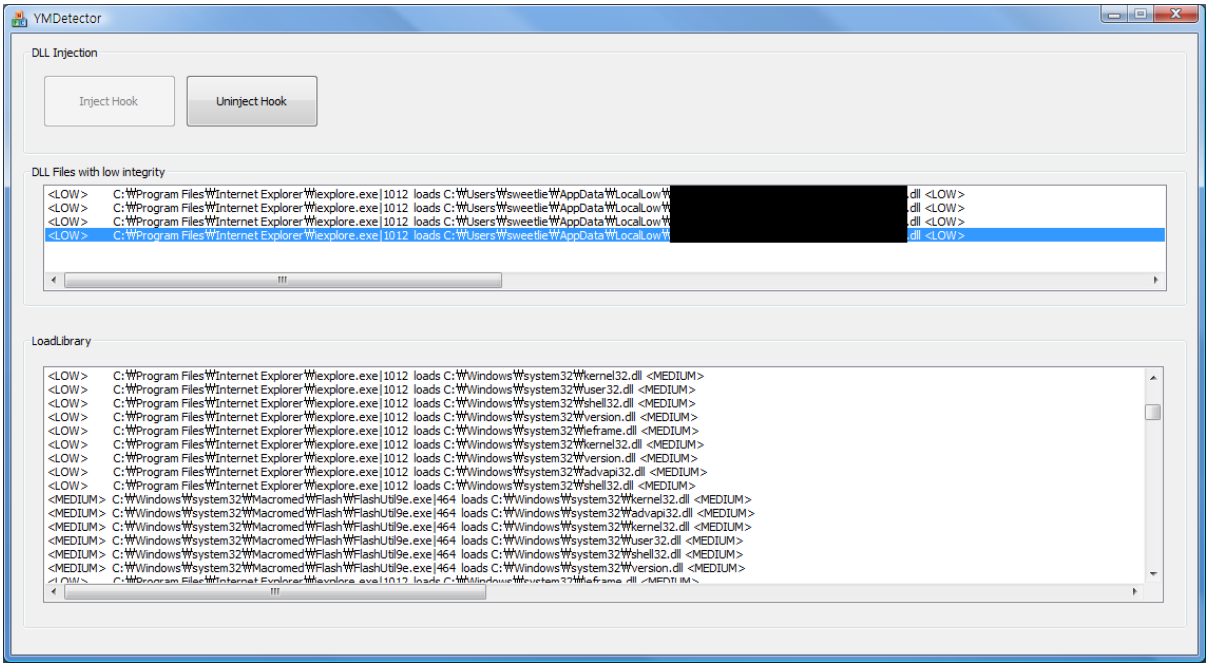

Figure 7. Tool to detect process loading DLL file with low integrity

Privilege Elevation of ActiveX control

Until now, we have discussed ActiveX control when privilege is not elevated. If ActiveX control with elevated privilege is vulnerable, a malicious user can obtain full control of victim's system. Therefore, ActiveX control with elevated privilege is more dangerous. We classify privilege elevation of ActiveX control into two groups, explicit and implicit, and describe details of each group.

1. Explicit Privilege Elevation of ActiveX control

Explicit privilege elevation of ActiveX control requires user agreement. In other words, when ActiveX control needs privilege elevation, user have to click "Continue" or "Allow" button on consent pop-ups. Figure 8 shows various types of consent pop-up.

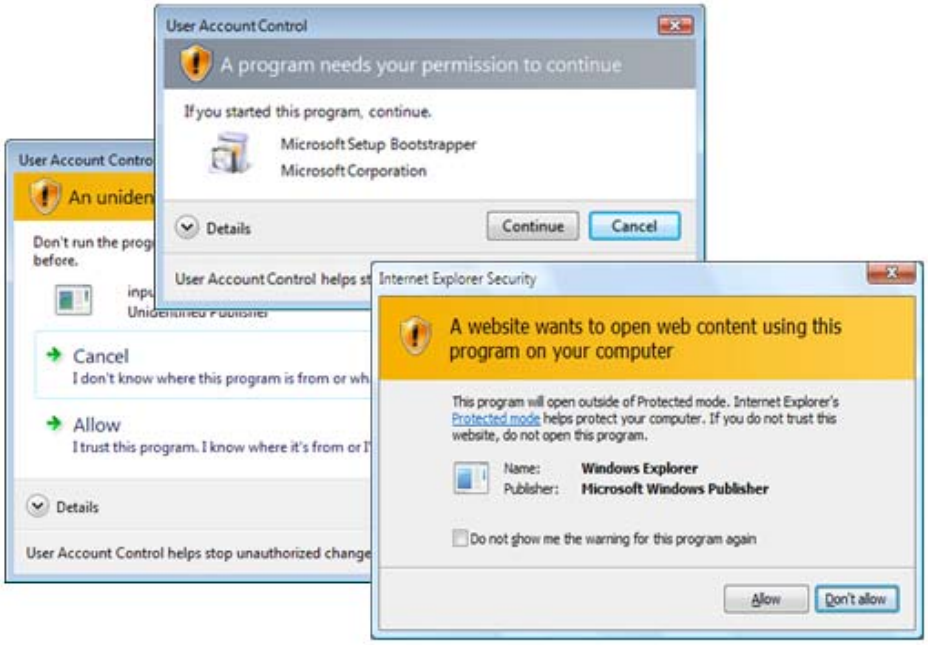

Figure 8. Various consent pop-ups

Microsoft provides CoCreateInstanceAsAdmin for explicit privilege elevation of ActiveX control. CoCreateInstanceAsAdmin launches ActiveX control at medium integrity and above after user agreement.

Another method of explicit privilege elevation is executing a higher-privileged surrogate process. User agreement is also required when ActiveX control tries to run the surrogate process by calling ShellExecute or CreateProcess.

Even if higher-privileged ActiveX control or surrogate processes have vulnerabilities, they cannot be exploited silently.

2. Implicit (Silent) Privilege Elevation of ActiveX control

Implicit privilege elevation allows ActiveX control to access higher-privileged resources without user agreement. There are several methods of implicit privilege elevation.

One is using elevation policy. This is used when ActiveX control needs medium integrity privilege. Microsoft provides the registry key for elevation policy, whose path is "HKLM₩SOFTWARE₩Microsoft₩Internet Explorer₩Low Rights₩ElevationPolicy₩{GUID of ActiveX control}₩". Figure 9 shows an example of the use of elevation policy. "AppName" and "AppPath" are for name and path of the surrogate process to be run by ActiveX control. "Policy" should be 3 to run a process silently at medium integrity.

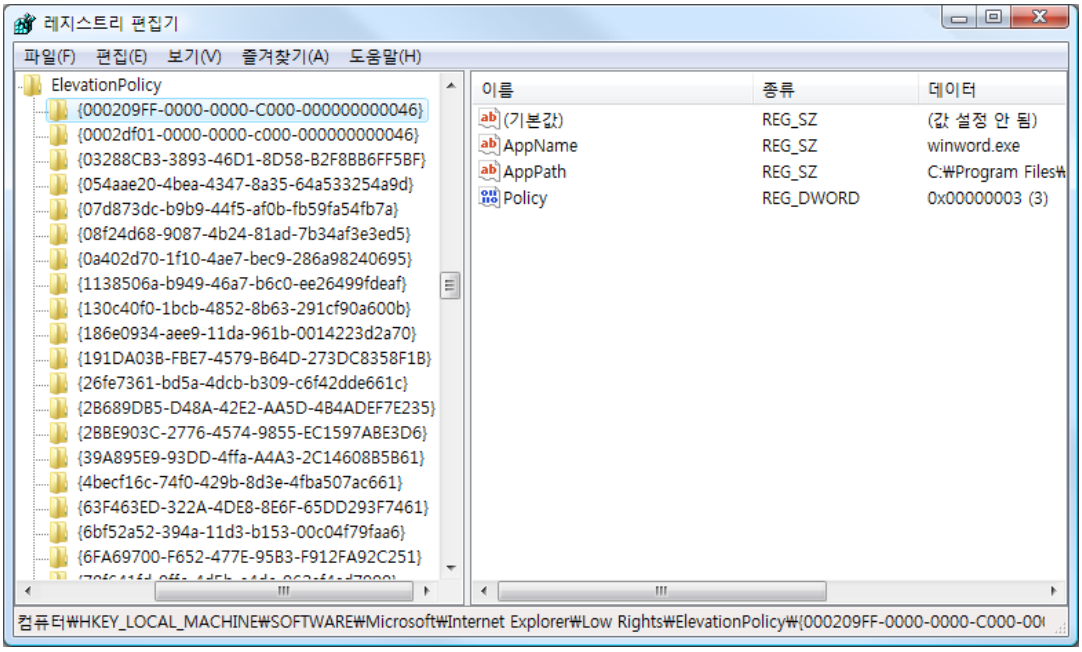

Figure 9. Example of use of elevation policy

Another method of privilege elevation is using a resident higher-privileged surrogate process. When ActiveX control is installed, a higher-privileged surrogate process can be registered as a startup program. Then, ActiveX control, by communication with the surrogate process, is able to access higher-privileged resources without user agreement. In other words, ActiveX control can elevate the restricted privilege by ordering the surrogate process. Figure 10 shows this scenario.

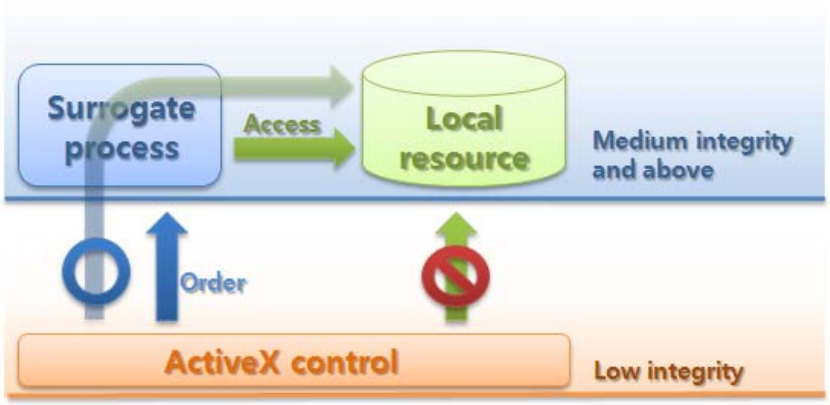

Figure 10. Implicit privilege elevation by surrogate process

Several types of communication can be possible with this model: files, registry keys, windows messages, named pipes, file mapping, and RPC (Remote Procedure Call). Some practical examples are as follows.

☐ Sharing files with low integrity

	Pseudo Code

ActiveX Control	…
	SHGetKnownFolderPath(&path); // Get the path with low integrity
	MyInstallFile("http://webserver/module.dll", path); // Download "module.dll" into the path
	…

Surrogate Process	…
	SHGetKnownFolderPath(&path); // Get the path with low integrity
	LoadLibrary(path+"module.dll"); // Load "module.dll"
	…

☐ Sharing registry keys with low integrity

	Pseudo Code

ActiveX Control	…
	IEGetWriteableHKCU(&key); // Get the registry key with low integrity
	MyWriteUpdateURL("http://webserver", key); // Write URL on the registry key
	…

Surrogate Process	…
	IEGetWriteableHKCU(&key); // Get the registry key with low integrity
	url = MyReadUpdateURL(key); // Read URL from the registry key
	MyUpdate(url); // Download updated files from URL
	…

☐ Windows Messages

	Pseudo Code
Surrogate Process	… ChangeWindowMessageFilter(WM_COPYDATA, MSGFLT_ADD); /* Allow to receive the WM_COPYDATA message from processes with low integrity WM_COPYDATA message is used to transmit a string */ …

ActiveX Control	…
	CString str = "htt://webserver/update.inf"; // Message to send to the surrogate process
	
	COPYDATASTRUCTURE cds;
	cds.cbData = str.GetLength()+1;
	cds.lpData = (LPSTR)(LPCSTR)str;
	
	// Get the window pointer of the surrogate process
	CWnd *pWnd = FindWindow([surrogate process name], NULL);

	// Send message
	SendMessage(pWnd->m_hWnd, WM_COPYDATA, (WPARAM)m_hWnd, (LPARAM)&cds);
	…

□ Named Pipes

	Pseudo Code

Surrogate Process	…
	PSECURITY_DESCRIPTOR pSD = NULL;
	PACL pSacl = NULL;
	BOOL fSaclPresent = FALSE;
	BOOL fSaclDefaulted = FALSE;
	
	// Create a named pipe
	HANDLE hPipe = CreateNamedPipe("₩₩₩₩.₩₩pipe₩₩sharedname", …);
	
	// Set SECURITY DESCRIPTOR and ACL to low integrity
	ConvertStringSecurityDescriptorToSecurityDescriptor("S:(ML;;NW;;;LW)", SDDL_REVISION_1, &pSD, NULL);
	GetSecurityDescriptorSacl(pSD, &fSaclPresent, &pSacl, &fSaclDefaulted);
	
	// Set the named pipe to low integrity
	SetSecurityInfo(hPipe, SE_KERNEL_OBJECT, LABEL_SECURITY_INFORMATION, NULL, NULL, NULL, pSacl);
	…

ActiveX Control	…
	DWORD dwBytesWritten;
	char buffer = [MessageToSend]; // Message to send to the surrogate process
	
	HANDLE hPipe = CreateFile("₩₩₩₩.₩₩pipe₩₩sharedname", …); // Open a named pipe
	WriteFile(hPipe, buffer, strlen(buffer), &dwBytesWritten, NULL); // Send the message
	…

□ File Mapping (surrogate process first)

	Pseudo Code
Surrogate Process	… unsigned char pszSecurity[SECURITY_DESCRIPTOR_MIN_LENGTH]; PSECURITY_DESCRIPTOR pSD = NULL; PACL pSacl = NULL; BOOL fSaclPresent = FALSE; BOOL fSaclDefaulted = FALSE; // Initialize SECURITY DESCRIPTOR & ACL

	InitializeSecurityDescriptor(pszSecurity,SECURITY_DESCRIPTOR_REVISION);
	SetSecurityDescriptorDacl(pszSecurity, TRUE, 0, FALSE);
	
	// Set SECURITY DESCRIPTOR and ACL to low integrity
	ConvertStringSecurityDescriptorToSecurityDescriptor("S:(ML;;NW;;;LW)", SDDL_REVISION_1, &pSD, NULL);
	GetSecurityDescriptorSacl(pSD, &fSaclPresent, &pSacl, &fSaclDefaulted);
	SetSecurityDescriptorSacl(pszSecurity, TRUE, pSacl, FALSE);
	
	// Create a memory mapped file with low integrity
	CreateFileMapping(INVALID_HANDLE_VALUE, pszSecurity, PAGE_READWRITE, 0, BUF_SIZE, "[sharedname]");
	…

ActiveX Control	…
	LPCTSTR pBuf;
	
	// Open the memmory mapped file
	HANDLE hMapFile = OpenFileMapping(FILE_MAP_WRITE, FALSE, "[sharedname]");
	pBuf = MapViewOfFile(hMapFile, FILE_MAP_WRITE, 0, 0, BUF_SIZE); // Get a buffer to write
	strcpy(pBuf, [MessageToSend]); // Send a message
	…

☐ File Mapping (ActiveX control first)

	Pseudo Code

ActiveX Control	…
	LPCTSTR pBuf;
	/*
	ActiveX control creates a memory mapped file with low integrity.
	*/
	// Create a memory mapped file
	HANDEL hMapFile = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, BUF_SIZE, "[sharedname]");
	pBuf = MapViewOfFile(hMapFile, FILE_MAP_WRITE, 0, 0, BUF_SIZE); // Get a buffer to write
	strcpy(pBuf, [MessageToSend]); // Send a message
	…

Surrogate Process	…
	LPCTSTR pBuf;
	
	// Open the memory mapped file
	HANDLE hMapFile = OpenFileMapping(FILE_MAP_READ, FALSE, "[sharedname]");
	pBuf = MapViewOfFile(hMapFile, FILE_MAP_READ, 0, 0, BUF_SIZE); // Get the buffer to read
	MyReadMapFile(); // Read the message
	…

□ RPC (with pipe)

	Pseudo Code

Surrogate Process	…
	unsigned char pszSecurity[SECURITY_DESCRIPTOR_MIN_LENGTH];
	PSECURITY_DESCRIPTOR pSD = NULL;
	PACL pSacl = NULL;
	BOOL fSaclPresent = FALSE;
	BOOL fSaclDefaulted = FALSE;
	
	// Initialize SECURITY DESCRIPTOR & ACL
	InitializeSecurityDescriptor(pszSecurity,SECURITY_DESCRIPTOR_REVISION);
	SetSecurityDescriptorDacl(pszSecurity, TRUE, 0, FALSE);
	
	// Set SECURITY DESCRIPTOR and ACL to low integrity
	ConvertStringSecurityDescriptorToSecurityDescriptor("S:(ML;;NW;;;LW)", SDDL_REVISION_1, &pSD, NULL);
	GetSecurityDescriptorSacl(pSD,&fSaclPresent, &pSacl, &fSaclDefaulted);
	SetSecurityDescriptorSacl(pszSecurity, TRUE, pSacl, FALSE);
	
	// Create an RPC server with low integrity
	RpcServerUseProtseqEp("ncacn_np", 20, "₩₩₩₩.₩₩pipe₩₩[sharedname]", pszSecurity);
	…

ActiveX Control	…
	// Connect the RPC server
	RpcStringBindingCompose(NULL, "ncacn_np", "localhost", "₩₩₩₩.₩₩pipe₩₩[sharedname]", …);
	MyCall([MessageToSend]); // Call an RPC function
	…

□ RPC (with TCP)

	Pseudo Code

Surrogate Process	…
	RpcServerUseProtseqEp("ncacn_ip_tcp", …, "[Port#]", NULL); // Create an RPC server
	…

ActiveX Control	…
	RpcStringBindingCompose(NULL, "ncacn_ip_tcp", "[Port#]", …); // Connect the RPC Server
	MyCall([MessageToSend]); // Call an RPC function
	….

Frequent consent pop-ups annoy users. For user convenience, developers want to minimize the number of consent pop-ups. Therefore, implicit privilege elevation is attractive because it doesn't require consent pop-ups.

However, implicit privilege elevation of ActiveX control may cause critical security threats. If ActiveX control with implicit privilege elevation has vulnerabilities, malicious users can obtain high privilege silently. Therefore, the implicit privilege elevation of ActiveX control has to be considered when ActiveX control on Windows Vista is inspected.

Conclusion

The most important change of ActiveX control on Windows Vista is that it cannot write files or keys in most restricted folders or registry keys. This prevents a malicious program from being executed repeatedly. However, this rule is no longer valid if some sensitive data are stored in a low integrity folder.

Therefore, developers should not install any program files in low integrity folders. They should not store any sensitive data in low integrity folders. In addition, they should obtain user agreement before elevating privilege of ActiveX control.

Related to ActiveX control security, Windows Vista will be the same as Windows XP if developers do not follow these rules.

Reference

1. Su Yong Kim, Do Hoon Lee, Sung Deok Cha, "Playing with ActiveX controls", CanSecWest 2007, http://cansecwest.com/csw07/csw07-suyongkim-dohoonlee.zip
2. Marc Silbey, Peter Brundrett, "Understanding and Working in Protected Mode Internet Explorer", MSDN, Sep. 2006
3. Microsoft Corporation, "Developer Best Practices and Guidelines for Applications in a Least Privileged Environment", Sep. 2005
4. Sharon Cohen, Rob Franco, "ActiveX Security: Improvements and Best Practices", MSDN, Sep. 2006
5. Microsoft Corporation, "Interprocess Communications", MSDN, http://msdn2.microsoft.com/en-us/liblary/aa365574.aspx
6. Chris Corio, "Teach Your Apps To Play Nicely With Windows Vista User Account Control", MSDN, http://msdn2.microsoft.com/en-us/magazine/cc163486.aspx, Jan. 2007
7. Michael Dunn, "A Developer's Survival Guide to IE Protected Mode", http://www.codeproject.com/KB/vista-security/PMSurvivalGuide.aspx, May. 2007
8. Microsoft Corporation, "Designing Applications to Run at a Low Integrity Level",

MSDN, http://msdn2.microsoft.com/en-us/liblary/bb625960.aspx
9. Microsoft Corporation, "CreateProcessAsUser Function", MSDN, http://msdn.microsoft.com/en-us/library/ms682429(VS.85).aspx
10. Microsoft Corporation, "Introduction to the Protected Mode API", MSDN, http://msdn.microsoft.com/en-us/library/ms537319.aspx
11. Microsoft Corporation, "Appendix A: SDDL for Mandatory Labels", MSDN, http://msdn.microsoft.com/en-us/library/bb625958.aspx