Binary Planting Goes "EXE" Yesterday, Apple issued new versions of the Safari browser that fix a binary planting vulnerability our company has reported to them in March this year under our then-effective disclosure policy. (See Apple's advisory .)

In the last 20 days since the binary planting monster escaped to the wilderness, eager bug-hunters were focused on unsafe loading of libraries, and understandably so: free tools were made available, and instructions were published on how to use monitoring software like Sysinternals' Process Monitor for detecting unsafe library loadings. As it turned out, tools + instructions + 20 days = 117 remotely exploitable vulnerabilities (at the time of this writing). The list is growing and will likely surpass our own list of 396 DLL planting and 127 EXE planting vulnerabilities at some time.

As already hinted by Acros , the search path problem does not only affect DLLs, or more accurately, LoadLibrary* functions. Indeed, this problem also affects the way Windows processes are launched via various functions such as CreateProcess* , ShellExecute* , WinExec , LoadModule , _spawn*p* and _exec*p* . (Asterisks denote various characters to avoid listing too many function names.) And if you thought the library-loading search path was dangerous, you'll probably be surprised about the (less documented) search paths for loading executables.

Let's revise the library-loading search path, as used by the LoadLibrary* functions.

LoadLibrary* search path

1. The directory from which the application loaded
2. 32-bit System directory (Windows\System32)
3. 16-bit System directory (Windows\System)
4. Windows directory (Windows)
5. Current working directory

6. Directories in the PATH environment variable

It is obvious from this search path that a binary planting vulnerability can only occur when the requested library - specified without a path - is not found in the first four locations. This means, and our research confirms it, that most DLL planting issues will result from trying to load a library that doesn't even exist on the system (with a few exceptions). This nicely explains what we dubbed the "dwmapi.dll phenomena": many Windows applications written for both XP and Vista/7 execute LoadLibrary("dwmapi.dll") to see if Vista Desktop Window Manager API is available. While on Vista/7 this DLL is found in system directories, it isn't on XP, leading to d
wmapi.dll
being loaded from the current working directory. Interestingly, our research has also found opposite cases where an application is only vulnerable on Vista or 7, but not on XP or older Windows systems.

Now let's look at the various search paths Windows are using for launching executables. In contrast to the LoadLibrary* search path, many of these aren't officially documented but can easily be observed with a file/process monitor.

CreateProcess*, WinExec and LoadModule search paths

1. The directory from which the application loaded
2. Current working directory
3. 32-bit System directory (Windows\System32)
4. 16-bit System directory (Windows\System)
5. Windows directory (Windows)
6. Directories in the PATH environment variable

This search path is identical to the old search path used by LoadLibrary* functions before the "safe search order" was introduced years ago. Apparently the current working directory is in the second place, which means that when an application tries to launch the Windows Calculator by calling something like
CreateProcess(NULL,"calc.exe",...)
, a malicious
calc.exe
lurking in the current working directory will get launched instead. And remotely, too, if the

current working directory happens to point to a remote network share in a local network or on Internet. And no, launching remote executables using these functions will never issue any security warnings to the user, in contrast to
ShellExecute*
. As far as we know, introducing
ShellExecute
-like security warnings to these functions would cause serious problems with various batch jobs and server back-end operations running without humans present.

ShellExecute* search path

1. Current working directory
2. 32-bit System directory (Windows\System32)
3. 16-bit System directory (Windows\System)
4. Windows directory (Windows)
5. Directories in the PATH environment variable
6. Directories specified in the App Paths registry key

Now this search path could not be friendlier to your binary planting attacker: the current working directory, possibly under attacker's control, is the first location consulted for the executable launched via a ShellExecute(NULL,"open","cmd.exe", ...) call. Fortunately though, Windows will issue a security warning when launching an executable from the Internet zone (but not from Local Intranet and My Computer zones). So for those of you who believe that users won't okay such warning the second time around after seeing that canceling it the first time failed to get them where they wanted to go, feel free to consider the
ShellExecute
binary planting vector intranet- and local-only.

_spawn*p* and _exec*p* search paths

1. Current working directory
2. 32-bit System directory (Windows\System32)
3. Windows directory (Windows)
4. Directories in the PATH environment variable

These functions, while probably used less frequently than CreateProcess* or ShellExecute*, provide a powerful binary planting potential: Trying to launch Notepad by calling
_execlp("notepad.exe","r")
will result in executing a malicious
notepad.exe
waiting in the current working directory, without a security warning under any circumstances.

There are many other ways to launch new processes in Windows, but most are likely to be wrappers for CreateProcess* or ShellExecute* functions, making their binary planting potential equal to that of their "wrapees".

So what have we learned today? First, binary planting is as much a problem with EXEs as it is with DLLs. Second, while DLL planting usually requires the DLL to be absent from the system, EXE planting works nicely with existing executables in system directories. And third, an application can be vulnerable depending on the version of Windows it is running on. Now make sure to test your applications on different Windows systems and update your Process Monitor filters to look for executables being loaded from the current working directory.

Are there any systemic countermeasures you can apply against EXE planting? Unfortunately not as many as for DLL planting:

- Microsoft's CWDIllegalInDllSearch hotfix only blocks the current working directory in LoadLibrary*
calls, so it can't protect you or your applications from EXE planting attacks. According to this guidance
, "Microsoft recommends that developers be careful when loading binaries and specify the fully qualified path name. This should pose less complexity when loading a binary as opposed to a library."

- The SetDllDirectory function also only affects LoadLibrary* calls. There is no SetExeDirectory
function available at this time to change the search path of
CreateProcess
,
ShellExecute
and other process-launching functions.

- Changing the current working directory to some safe location either at the beginning of the application's execution or before loading libraries and launching processes can limit the exploitability of both DLL and EXE planting vulnerabilities. (However, applications tend to change their current working directory even if the developer had no intention for them to do so; but more about this in another blog post.)