Using delayload to detect functionality is a security vulnerability

devblogs.microsoft.com/oldnewthing/20101111-00

November 11, 2010

Raymond Chen

We saw last time that your debugging code can be a security vulnerability when you don't control the current directory. A corollary to this is that your delayload code can also be a security vulnerability, for the same reason. When you use the linker's delayload functionality to defer loading a DLL until the first time it is called, the linker injects code which calls `LoadLibrary` on a DLL the first time you call a function in it, and then calls `GetProcAddress` on the functions you requested. When you call a delay-loaded function and the delayload code did not get a function pointer from `GetProcAddress` (either because the DLL got loaded but the function does not exist, or because the DLL never got loaded in the first place), it raises a special exception indicating that a delayed load failed. Let's look again at the order in which the `LoadLibrary` function searches for a library:

1. The directory containing the application EXE.
2. The system32 directory.
3. The system directory.
4. The Windows directory.
5. The current directory.
6. The PATH.

The code which implements the delayload functionality uses a relative path when it passes the library name to `LoadLibrary`. (It has no choice since all it has to work with is the library name stored in the IMPLIB.) Consequently, if the DLL you are delay-loading does not exist in any of the first four search locations, the `LoadLibrary` will look in location 5: the current directory. At this point, the current directory attack becomes active, and a bad guy can inject an attack DLL into your process. For example, this sample code uses delayload to detect whether the functions in `dwmapi.dll` exist, calling them if so. If the function `IsThemeEnabled` is not available, then it treats themes as not enabled. If the program runs on a system without `dwmapi.dll`, then the delayload will throw an exception, and the exception is caught and turned into a failure. Disaster avoided. But in fact, the disaster was not avoided; it was *introduced*. If you run the program on a system without `dwmapi.dll`, then a bad guy can put a rogue copy of `dwmapi.dll` into the current directory, and *boom* your process just loaded an untrusted DLL. Game over. Using the delayload feature to probe for a DLL is morally equivalent to using a plain `LoadLibrary` to probe for the presence of a debugging DLL. In both cases, you are looking for a DLL with the expectation that there's a good chance it won't be there. But it is exactly in those *sometimes it won't be there* cases where you become vulnerable to attack. If you want to probe for the existence of a DLL, then you need to know what directory the DLL *should* be in, and then load that DLL via that full path in order to avoid the current directory attack. On the other hand, if the DLL you want to delayload is known to be installed in a directory ahead of the current directory in the search path (for example, you require versions of the the operating system in which the DLL is part of the mandatory install, and the directory in which it is installed is the System32 directory) then you can use delayload. In other words, you can use delayload for delaying the load of a DLL. But if you're using delayload to probe for a DLL's existence, then you become vulnerable to a current directory attack.

This is one of those subtle unintended consequences of changing the list of files included with an operating system. If you take what used to be a mandatory component that can't be uninstalled, and you change it to an optional component that can be uninstalled, then not only do programs which linked to the DLL in the traditional manner stop loading, but you've also *introduced a security vulnerability*: Programs which had used delayload under the calculation (correct at the time it was made) that doing so was safe are now vulnerable to the current directory attack.

Raymond Chen

Follow