Your debugging code can be a security vulnerability: Loading optional debugging DLLs without a full path

devblogs.microsoft.com/oldnewthing/20101110-00

November 10, 2010

Raymond Chen

Remember, the bad guys don't care that your feature exists just for debugging purposes. If it's there, they will attack it.

Consider the following code:

```
DOCLOADINGPROC g_pfnOnDocLoading;
void LoadDebuggingHooks()
{
 HMODULE hmodDebug = LoadLibrary(TEXT("DebugHooks.dll"));
 if (!hmodDebug) return;
 g_pfnOnDocLoading = (DOCLOADINGPROC)
               GetProcAddress(hmodDebug, "OnDocLoading");
 ...
}
HRESULT LoadDocument(...)
{
 ...
 if (g_pfnOnDocLoading) {
   // let the debugging hook replace the stream
   g_pfnOnDocLoading(&pstmDoc);
 }
 ...
}
```

When you need to debug the program, you can install the `DebugHooks.dll` DLL into the application directory. The code above looks for that DLL and if present, gets some function pointers from it. For illustrative purposes, I've included one debugging hook. The idea of this example (and it's just an example, so let's not argue about whether it's a good example) is that when we're about to load a document, we call the `OnDocLoading` function, telling it about the document that was just loaded. The `OnDocLoading` function wraps the `IStream` inside another object so that the contents of the document can be logged byte-by-byte as it is loaded, in an attempt to narrow down exactly where document loading fails. Or it can be used for testing purposes to inject I/O errors into the document loading path to confirm that the program behaves properly under those conditions. Use your imagination.

But this debugging code is also a security vulnerability.

Recall that the library search path searches directories in the following order:

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.

When debugging your program, you install `DebugHooks.dll` into the application directory so that it is found in step 1. But when your program isn't being debugged, the search in step 1 fails, and the search continues in the other directories. The DLL is not found in steps 2 through 4, and then we reach step 5: The current directory.

And now you're pwned.

Your application typically does not have direct control over the current directory. The user can run your program from any directory, and that directory ends up as your current directory. And then your `LoadLibrary` call searches the current directory, and if a bad guy put a rogue DLL in the current directory, your program just becames the victim of code injection.

This is made particularly dangerous when your application is associated with a file type, because the user can run your application just by double-clicking an associated document.

When you double-click a document, Explorer sets the current directory of the document handler application to the directory that contains the document being opened. This is necessary for applications which look around in the current directory for supporting files. For example, consider a hypothetical application `LitWare Writer` associated with `*.LIT` files. A LitWare Writer document `ABC.LIT` file is really just the representative for a family of files, `ABC.LIT` (the main document), `ABC.LTC` (the document index and table of contents), `ABC.LDC` (the custom spell check dictionary for the document), `ABC.LLT` (the custom document layout template), and so on. When you open the document `C:\PROPOSAL\ABC.LIT`, LitWare Writer looks for the other parts of your document in the current directory, rather than in `C:\PROPOSAL`. To help these applications find their files, Explorer specifies to the `CreateProcess` function that it should set the initial current directory of LitWare Writer to `C:\PROPOSAL`.

Now, you might argue that programs like LitWare Writer (which look for the ancillary files of a multi-file document in the current directory instead of the directory containing the primary file of the multi-file document) are poorly-written, and I would agree with you, but Windows needs to work even with poorly-written programs. (**Pre-emptive snarky comment**:

Windows is itself a poorly-written program.) There are a lot of poorly-written programs out there, some of them industry leaders in their market (**see above pre-emptive snarky comment**) and if Windows stopped accommodating them, people would say it was the fault of Windows and not the programs.

I can even see in my mind's eye the bug report that resulted in this behavior being added to the MS-DOS Executive:

"This program has worked just fine in MS-DOS, but in Windows, it doesn't work. Stupid Windows."

Customers tend not to be happy with the reply, "Actually, that program has simply been lucky for the past X years. The authors of the program never considered the case where the document being opened is not in the current directory. And it got away with it, because the way you opened the document was to use the `chdir` command to move to the directory that contained your document, and then to type `LWW ABC.LIT`. If you had ever done `LWW C:\PROPOSAL\ABC.LIT` you would have run into the same problem. The behavior is by design."

In response to "The behavior is by design" is usually "Well, a design that prevents me from getting my work done is a crappy design." or a much terser "No it's not, it's a bug." (Don't believe me? Just read Slashdot.)

So to make these programs work in spite of themselves, the MS-DOS Executive sets the current directory of the program being launched to the directory containing the document itself. This was not an unreasonable decision because it gets the program working again, and it's not like the program cared about the current directory it inherited from the MS-DOS Executive, since it had no control over that either!

But it means that if you launched a program by double-clicking an associated document, then unless that program takes steps to change its current directory, it will have the document's containing folder as its current directory, which prevents you from deleting that directory.

Bonus chatter: I wrote this series of entries nearly two years ago, and even then, I didn't consider this to be anything particularly groundbreaking, but apparently some people rediscovered it a few months ago and are falling all over themselves to claim credit for having found it first. It's like a new generations of teenagers who think they invented sex. For the record, here is some official guidance. (And just to be clear, that's official guidance on the current directory attack, not official guidance on sex.)

History chatter: Why is the current directory even considered at all? The answer goes back to CP/M. In CP/M, there was no PATH. Everything executed from the current directory. The rest is a chain of backward compatibility.

Raymond Chen

Follow