It rather involved being on the other side of this airtight hatchway: Writing to the application directory

devblogs.microsoft.com/oldnewthing/20121207-00

December 7, 2012

Raymond Chen

We received a security vulnerability report that went roughly like this:

> There is a security vulnerability in the X component. It loads `shell32.dll` from the current directory, thereby making it vulnerable to a current directory attack. Here is a sample program that illustrates the problem. Copy a rogue `shell32.dll` into the current directory and run the program. Observe that the rogue `shell32.dll` is loaded instead of the system one.

If you actually followed the instructions, what you saw depended on your definition of "run the program." Let's assume that the program has been placed in the directory `C:\sample\sample.exe`.

1. Setting the current directory to the application directory.

   ```
   cd /d C:\sample
   copy \\rogue\server\shell32.dll
   c:\sample\sample.exe
   ```

   In this case, the attack succeeds.

2. Setting the current directory to an unrelated directory.

   ```
   cd /d %USERPROFILE%
   copy \\rogue\server\shell32.dll
   c:\sample\sample.exe
   ```

   In this case, the attack fails.

3. Running the application from Explorer.

   ```
   copy \\rogue\server\shell32.dll C:\sample
   double-click sample.exe in Explorer
   ```

   In this case, the attack succeeds.

Let's look at case 3 first. In case 3, what is the current directory? When you launch a program from Explorer, the current directory is set to the directory of the thing you double-clicked. Therefore, case 3 is identical to case 1. That's one less case to have to study.

We also see that the attack is not strictly a current directory attack, because the attack failed in case 2 even though a rogue `shell32.dll` was in the current directory.

What we're actually seeing is an *application directory* attack.

Recall that the application directory is searched ahead of the system directory. Therefore, you can override a file in the system directory by putting it in your application directory. This is part of the *directory as a bundle* principle. If you packaged a DLL with your application, then presumably that's the one you want, even if a future version of Windows decides to create a DLL of the same name.

The vulnerability report sort of acknowledged that this was an application directory attack rather than a current directory attack when they explained why this is a serious problem:

> By placing a rogue copy of `shell32.dll` in the `C:\Program Files\Microsoft Office\Office12` directory, an attacker can inject arbitrary code into all Office applications.

If the attack were really a current directory attack, the attacker would have put a rogue copy of `shell32.dll` in the directory containing your Excel spreadsheet, not the directory containing `EXCEL.EXE`.

And that's where you reach the airtight hatchway: Normal users do not have write permission into the `C:\Program Files\Microsoft Office\Office12` directory. You need administrator privileges to create files there. And if you have administrator privileges, then you already pwn the machine. It's not really a vulnerability that you can do anything you want once you pwn the machine.

Of course, this non-vulnerability does expose a security issue you need to bear in mind when you run your own programs: Your application's directory is its airtight hatchway. Make sure you control who you let in! If you leave your application directory world-writeable, then you've effectively left your airtight hatchway unlocked. This is one reason why the Microsoft Logo guidelines recommend (require?) that programs be installed into the Program Files directory: The default security descriptor for subdirectories of Program Files does not grant write permission to normal users. It's secure by default.

There are many variations of this type of vulnerability report, and they nearly always are mischaracterized as a current directory attack. They usually go like this:

> There is a DLL planting vulnerability in LITWARE.EXE. Place a rogue DLL named SHELL32.DLL in the same directory as LITWARE.EXE. When LITWARE.EXE is run, the rogue DLL is loaded from the current directory, resulting in code injection.

The person who submits the report has confused the application directory with the current directory, probably because they never considered that the two might be different.

```
C:\> mkdir C:\test
C:\> cd C:\test
C:\test> copy \\trusted\server\LITWARE.EXE
C:\test> copy \\rogue\server\SHELL32.DLL
C:\test> LITWARE
-- observe that the rogue DLL is loaded
-- proof of current directory attack
```

They never tried this:

```
C:\> mkdir C:\test
C:\> cd C:\test
C:\test> copy \\trusted\server\LITWARE.EXE
C:\> mkdir C:\test2
C:\> cd C:\test2
C:\test2> copy \\rogue\server\SHELL32.DLL
C:\test2> ..\test\LITWARE
-- observe that the rogue DLL is not loaded
```

That second experiment shows that the attack is not a current directory attack at all. It's an application directory attack.

Each time one of these reports comes in, we have to perform the same evaluation to confirm that it really is an application directory attack and not a current directory attack. (This means, among other things, repeating the test on every version of Windows, and every version of LitWare, and every combination of the two, just to make sure all the possibilities have been covered. The odds are strong that it will all turn into a false alarm, but who knows. Maybe there's something about the interaction between LitWare 5.2 SP2 and Windows XP SP3 that triggers a new code path that does indeed try to load `shell32.dll` from the current directory. And it's that specific combination of circumstances the person was trying to report, but did a bad job of expressing.)

Raymond Chen

Follow