Dynamic-link library redirection - Win32 apps

By stevewhims

Archived: 2026-04-06 00:04:40 UTC

The *DLL loader* is the part of the operating system (OS) that resolves references to DLLs, loads them, and links them. Dynamic-link library (DLL) redirection is one of the techniques by which you can influence the behavior of the *DLL loader*, and control which one of several candidate DLLs it actually loads.

Other names for this feature include *.local*, *Dot Local*, *DotLocal*, and *Dot Local Debugging*.

If your application depends on a specific version of a shared DLL, and another application is installed with a newer or older version of that DLL, then that can cause compatibility problems and instability; it can cause your app to start to fail.

The DLL loader looks in the folder that the calling process was loaded from (the executable's folder) before it looks in other file system locations. So one workaround is to install the DLL that your app needs in your executable's folder. That effectively makes the DLL private.

But that doesn't solve the problem for COM. Two incompatible versions of a COM server can be installed and registered (even in different file system locations), but there's only one place to register the COM server. So only the latest registered COM server will be activated.

You can use redirection to solve these problems.

The rules that the DLL loader follows ensure that system DLLs are loaded from the Windows system locations—for example, the system folder (`%SystemRoot%\system32`). Those rules avoid planting attacks: where an adversary puts code that they wrote in a location that they can write to, and then convince some process to load and execute it. But the loader's rules also make it more difficult to work on OS components, because to run them requires updating the system; and that's a very impactful change.

But you can use redirection to load private copies of DLLs (for purposes such as testing, or measuring the performance impact of a code change).

If you want to contribute to the source code in the public WindowsAppSDK GitHub repository, then you'll want to test your changes. And, again, that's a scenario for which you can use redirection to load your private copies of DLLs instead of the versions that ship with the Windows App SDK.

In fact, there are two ways to ensure that your app uses the version of the DLL that you want it to:

- DLL redirection. Continue to read this topic for more details.
- Side-by-side components. Described in the topic Isolated applications and side-by-side assemblies.

Tip

If you're a developer or an administrator, then you should use DLL redirection for existing applications. That's because it doesn't require any changes to the app itself. But if you're creating a new app, or updating an existing app, and you want to isolate your app from potential problems, then create a side-by-side component.

To enable DLL redirection machine-wide, you must create a new registry value. Under the key `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options`, create a new **DWORD** value with the name *DevOverrideEnable*. Set the value to 1, and restart your computer. Or use the command below (and restart your computer).

```
reg add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options" /v DevOverrideEnable /
```

With that registry value set, DotLocal DLL redirection is respected even if the app has an application manifest.

To use DLL redirection, you'll create a *redirection file* or a *redirection folder* (depending on the kind of app you have), as we'll see in later sections in this topic.

A packaged app requires a special folder structure for DLL redirection. The following path is where the loader will look when redirection is enabled:

```
<Drive>:\<path_to_package>\microsoft.system.package.metadata\application.local\
```

If you're able to edit your `.vcxproj` file, then a convenient way to cause that special folder to be created and deployed with your package is to add some extra steps to the build in your `.vcxproj`:

```
 <ItemDefinitionGroup>
    <PreBuildEvent>
        <Command>
            del $(FinalAppxManifestName) 2&gt;nul
            <!-- [[Using_.local_(DotLocal)_with_a_packaged_app]] This makes the extra DLL deployed via F5 get lo
            if NOT EXIST $(IntDir)\microsoft.system.package.metadata\application.local MKDIR $(IntDir)\microsoft
            if EXIST "&lt;A.dll&gt;" copy /y "&lt;A.dll&gt;" $(IntDir)\microsoft.system.package.metadata\applica
            if EXIST "&lt;B.dll&gt;" copy /y "&lt;B.dll&gt;" $(IntDir)\microsoft.system.package.metadata\applica
        </Command>
    </PreBuildEvent>
</ItemDefinitionGroup>
<ItemGroup>
    <!-- Include any locally built system experience -->
    <Media Include="$(IntDir)\microsoft.system.package.metadata\application.local\**">
        <Link>microsoft.system.package.metadata\application.local</Link>
    </Media>
</ItemGroup>
```

Let's walk through some of what that configuration does.

1. Set up a `PreBuildEvent` for your Visual Studio **Start Without Debugging** (or **Start Debugging**) experience.

```
 <ItemDefinitionGroup>
  <PreBuildEvent>
```

2. Ensure that you have the correct folder structure in your intermediate directory.

```
 <!-- [[Using_.local_(DotLocal)_with_modern_apps]] This makes the extra DLL deployed via Start get loade
if NOT EXIST $(IntDir)\microsoft.system.package.metadata\application.local MKDIR $(IntDir)\microsoft.syst
```

3. Copy any DLLs that you've built locally (and wish to use in preference to the system-deployed DLLs) into the `application.local` directory. You can pick up DLLs from just about anywhere (we recommended that you use the available macros for your `.vcxproj`). Just be sure that those DLLs build before this project does; otherwise they'll be missing. Two *template* copy commands are shown here; use as many as you need, and edit the `<path-to-local-dll>` placeholders.

```
    if EXIST "<path-to-local-dll>" copy /y "<path-to-local-dll>" $(IntDir)\microsoft.system.package.metada
   if EXIST "<path-to-local-dll>" copy /y "<path-to-local-dll>" $(IntDir)\microsoft.system.package.metadat
   </Command>
</PreBuildEvent>
```

4. Lastly, indicate that you want to include the special directory and its contents in the deployed package.

```
 <ItemGroup>
  <!-- Include any locally built system experience -->
  <Media Include="$(IntDir)\microsoft.system.package.metadata\application.local\**">
    <Link>microsoft.system.package.metadata\application.local</Link>
  </Media>
</ItemGroup>
```

The approach described here (using an intermediate directory) keeps your source code control enlistment clean, and reduces the possibility of accidentally committing a compiled binary.

Next, all you need to do is to (re)deploy your project. In order to get a clean, full (re)deployment, you might also have to uninstall/clean out the existing deployment on your target device.

If you're not able to use your `.vcxproj` in the way shown above, then you can achieve the same ends by hand on your target device with a couple of simple steps.

1. Determine the package's installation folder. You can do that in PowerShell by issuing the command `Get-AppxPackage`, and looking for the *InstallLocation* that's returned.

2. Use that *InstallLocation* to change the ACLs to allow yourself to create folders/copy files. Edit the `<InstallLocation>` placeholders in this script, and run the script:

```
 cd <InstallLocation>\Microsoft.system.package.metadata
takeown /F . /A
icacls  . /grant Administrators:F
md <InstallLocation>\Microsoft.system.package.metadata\application.local
```

3. Lastly, manually copy any DLLs that you've built locally (and wish to use in preference to the system-deployed DLLs) into the `application.local` directory, and [re]start the app.

To confirm that the correct DLL is being loaded at runtime, you can use Visual Studio with the debugger attached.

1. Open the **Modules** window (**Debug** > **Windows** > **Modules**).
2. Find the DLL, and make sure that the **Path** indicates the redirected copy, and not the system-deployed version.
3. Confirm that only one copy of a given DLL is loaded.

The redirection file must be named `<your_app_name>.local`. So if your app's name is `Editor.exe`, then name your redirection file `Editor.exe.local`. You must install the redirection file in the executable's folder. You must also install the DLLs in the executable's folder.

The *contents* of a redirection file are ignored; its presence alone causes the DLL loader to check the executable's folder first whenever it loads a DLL. To mitigate the COM problem, that redirection applies both to full path and partial name loading. So redirection happens in the COM case and also regardless of the path specified to **LoadLibrary** or **LoadLibraryEx**. If the DLL isn't found in the executable's folder, then loading follows its usual search order. For example, if the app `C:\myapp\myapp.exe` calls **LoadLibrary** using the following path:

`C:\Program Files\Common Files\System\mydll.dll`

And if both `C:\myapp\myapp.exe.local` and `C:\myapp\mydll.dll` exist, then **LoadLibrary** loads `C:\myapp\mydll.dll`. Otherwise, **LoadLibrary** loads `C:\Program Files\Common Files\System\mydll.dll`.

Alternatively, if a folder named `C:\myapp\myapp.exe.local` exists, and it contains `mydll.dll`, then **LoadLibrary** loads `C:\myapp\myapp.exe.local\mydll.dll`.

If you're using DLL redirection, and the app doesn't have access to all drives and directories in the search order, then **LoadLibrary** stops searching as soon as access is denied. If you're *not* using DLL redirection, then **LoadLibrary** skips directories that it can't access, and then it continues searching.

It's good practice to install app DLLs in the same folder that contains the app; even if you're not using DLL redirection. That ensures that installing the app doesn't overwrite other copies of the DLL (thereby causing other apps to fail). Also, if you follow this good practice, then other apps don't overwrite your copy of the DLL (and don't cause your app to fail).
