Supplemental reading for Windows Package Dependencies

DLL Files and Windows Package Dependencies

In this reading, you will learn about dynamic link library (DLL) files. This information includes how Windows package dependencies can break and how Microsoft has remedied these DLL dependency problems using the .NET framework and other methods. You will also learn about the side-by-side assemblies and manifest files for Windows applications.

Dynamic link library (DLL)

Windows DLL files are vital to the core functions of the Windows operating system (OS). Some Windows-compatible applications also use DLL files to function. DLLs are made up of programming modules that contain reusable code. Multiple applications can use and reuse the same DLL files. For example, the Comdlg32 DLL file is used by many applications to provide Windows dialog box functions. The reusable feature helps Windows conserve disk space and use RAM more efficiently, which improves the operating speed of the OS and applications. The modular structure also makes updating a DLL file fast and simple, eliminating the need to update the entire library. DLL updates are installed once for use by any number of applications.

A few common DLLs used by Windows include:

- **.drv files** - Device drivers manage the operation of physical devices such as printers.
- **.ocx files** - Active X controls provide controls like the program object for selecting a date from a calendar.
- **.cpl files** - Control panel files manage each of the functions found in the Windows Control Panel.

An application can use DLLs to load parts of the app as modules. This means that if the application offers multiple functions, the app can selectively load only the modules that offer the functionality requested by the user. For example, if a user does not access the Print function within an application, then the printer driver DLL file does not need to be loaded into memory. This system requires less RAM to hold the application in working memory, which improves operating speeds.

DLL dependencies

A Windows package dependency is created when an application uses a DLL file. Although the Windows DLL system supports the sharing of DLL files by multiple applications, the applications' dependencies can be broken under certain circumstances.

DLL dependencies can be broken when:

- **Overwriting DLL dependencies** - It is possible for an application to overwrite the DLL dependency of another app, causing the other app to fail.
- **Deleting DLL files** - Some applications and malware may delete the DLLs needed by other applications installed on a system.
- **Applying upgrades or fixes to DLLs** - Can cause a problem called "DLL hell" where an application installs a new version of the shared DLL for a computer system. However, other applications that are dependent on the shared DLL have not yet been updated to be compatible with the new version of the DLL. This causes the other applications to fail when the end user tries to launch them.
- **Rolling-back to previous DLL versions** - A user may try to reinstall an older application that stopped working after a shared DLL file was upgraded by a newer app. However, the reinstallation of the app that uses the old DLL version can overwrite the new DLL file. This DLL version roll-back can cause the newer app with the shared DLL dependency to fail the next time it tries to run.

Microsoft has remedied these problems through the use of:

- **Windows File Protection -** The Windows OS controls the updates and deletions of system DLL files. Windows File Protection will allow only applications with valid digital signatures to update and delete DLL files.
- **Private DLLs** - Removes the sharing option from DLLs by creating a private version of the DLL and storing it in the application's root folder. Changes to the shared version of the DLL will not affect the application's private copy.
- **.NET Framework assembly versioning** - Resolves the "DLL hell" problem by allowing an application to add an updated version of a DLL file without removing the older version of the DLL file. This prevents the malfunction of applications that have dependencies on the older DLL file. The DLL versions can be found in the "C:\Windows\assembly" path and are placed in the Global Assembly Cache (GAC). The GAC contains the .NET "Strong Name Assembly" of each DLL file version. This "Strong Name Assembly" includes the:
    - **name of the assembly** - multiple DLL files can share the assembly name
    - **version number** - differentiates the version of DLLs
    - **culture** - country or region where the application is deployed, can be "neutral"
    - **public key token** - a unique 16-character key assigned to an assembly when it is built

Side-by-side assemblies

DLLs and dependencies can also be located in side-by-side assemblies. A side-by-side assembly is a public or private resource collection that is available to applications during run time. Side-by-side assemblies contain XML files called manifests. The manifests contain data similar to the configuration settings and other data that applications traditionally stored in the Windows registry. Instead of registering this data in the Windows registry, the applications store shared side-by-side assembly manifests in the WinSxS folder of the computer. Private manifests are stored inside the application's folder or they can be embedded in an application or assembly. The metadata of a manifest may include:

- **Names** - Manages file naming.
- **Resource collections** - Can include one or more DLLs, COM servers, Windows classes, interfaces, and/or type libraries.
- **Classes** - Included if versioning is used.
- **Dependencies** - Applications and assemblies can create dependencies to other side-by-side assemblies.

As an IT Support professional, this concept should be considered when troubleshooting application issues. If the application's configuration settings are not found in the Windows registry, they might be located in the manifest from the app's side-by-side assembly.

Mark as completed

Like **Dislike** **Report an issue**