

# Dynamic-Link Libraries (Dynamic-Link Libraries)

Article • 06/01/2022

A *dynamic-link library* (DLL) is a module that contains functions and data that can be used by another module (application or DLL).

A DLL can define two kinds of functions: exported and internal. The exported functions are intended to be called by other modules, as well as from within the DLL where they are defined. Internal functions are typically intended to be called only from within the DLL where they are defined. Although a DLL can export data, its data is generally used only by its functions. However, there is nothing to prevent another module from reading or writing that address.

DLLs provide a way to modularize applications so that their functionality can be updated and reused more easily. DLLs also help reduce memory overhead when several applications use the same functionality at the same time, because although each application receives its own copy of the DLL data, the applications share the DLL code.

The Windows application programming interface (API) is implemented as a set of DLLs, so any process that uses the Windows API uses dynamic linking.

- [About Dynamic-Link Libraries](#)
- [Using Dynamic-Link Libraries](#)
- [Dynamic-Link Library Reference](#)

## ⓘ Note

If you are a user experiencing difficulty with a DLL on your computer, you should contact customer support for the software vendor that publishes the DLL. If you feel you are in need of support for a Microsoft product (including Windows), please go to our technical support site at [support.microsoft.com](https://support.microsoft.com) <sup>↗</sup>.

# About Dynamic-Link Libraries

Article • 01/08/2021

Dynamic linking allows a module to include only the information needed to locate an exported DLL function at load time or run time. Dynamic linking differs from the more familiar static linking, in which the linker copies a library function's code into each module that calls it.

## Types of Dynamic Linking

There are two methods for calling a function in a DLL:

- In *load-time dynamic linking*, a module makes explicit calls to exported DLL functions as if they were local functions. This requires you to link the module with the import library for the DLL that contains the functions. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded.
- In *run-time dynamic linking*, a module uses the [LoadLibrary](#) or [LoadLibraryEx](#) function to load the DLL at run time. After the DLL is loaded, the module calls the [GetProcAddress](#) function to get the addresses of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by [GetProcAddress](#). This eliminates the need for an import library.

## DLLs and Memory Management

Every process that loads the DLL maps it into its virtual address space. After the process loads the DLL into its virtual address, it can call the exported DLL functions.

The system maintains a per-process reference count for each DLL. When a thread loads the DLL, the reference count is incremented by one. When the process terminates, or when the reference count becomes zero (run-time dynamic linking only), the DLL is unloaded from the virtual address space of the process.

Like any other function, an exported DLL function runs in the context of the thread that calls it. Therefore, the following conditions apply:

- The threads of the process that called the DLL can use handles opened by a DLL function. Similarly, handles opened by any thread of the calling process can be used in the DLL function.
- The DLL uses the stack of the calling thread and the virtual address space of the calling process.
- The DLL allocates memory from the virtual address space of the calling process.

For more information about DLLs, see the following topics:

- [Advantages of Dynamic Linking](#)
- [Dynamic-Link Library Creation](#)
- [Dynamic-Link Library Entry-Point Function](#)
- [Load-Time Dynamic Linking](#)
- [Run-Time Dynamic Linking](#)
- [Dynamic-Link Library Search Order](#)
- [Dynamic-Link Library Data](#)
- [Dynamic-Link Library Redirection](#)
- [Dynamic-Link Library Updates](#)
- [Dynamic-Link Library Security](#)
- [Applnit DLLs and Secure Boot](#)

---

## Feedback

# Advantages of Dynamic Linking

Article • 01/08/2021

Dynamic linking has the following advantages over static linking:

- Multiple processes that load the same DLL at the same base address share a single copy of the DLL in physical memory. Doing this saves system memory and reduces swapping.
- When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments, calling conventions, and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
- A DLL can provide after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was initially shipped.
- Programs written in different programming languages can call the same DLL function as long as the programs follow the same calling convention that the function uses. The calling convention (such as C, Pascal, or standard call) controls the order in which the calling function must push the arguments onto the stack, whether the function or the calling function is responsible for cleaning up the stack, and whether any arguments are passed in registers. For more information, see the documentation included with your compiler.

A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module. The system terminates processes using load-time dynamic linking if they require a DLL that is not found at process startup and gives an error message to the user. The system does not terminate a process using run-time dynamic linking in this situation, but functions exported by the missing DLL are not available to the program.

# Dynamic-link library creation

Article • 01/27/2024

To create a Dynamic-link library (DLL), you must create one or more source code files, and possibly a linker file for exporting the functions. If you plan to allow applications that use your DLL to use load-time dynamic linking, then you must also create an import library.

## Creating source files

The source files for a DLL contain exported functions and data, internal functions and data, and an optional [entry-point function](#) for the DLL. You may use any development tools that support the creation of Windows-based DLLs.

If your DLL may be used by a multithreaded application, then you should make your DLL "thread-safe". To avoid data corruption, you must synchronize access to all of the DLL's global data. You must also ensure that you link only with libraries that are thread-safe as well. For example, Microsoft Visual C++ contains multiple versions of the C Run-time Library, one that is not thread-safe and two that are.

## Exporting functions

How you specify which functions in a DLL should be exported depends on the tools that you are using for development. Some compilers allow you to export a function directly in the source code by using a modifier in the function declaration. Other times, you must specify exports in a file that you pass to the linker.

For example, using Visual C++, there are two possible ways to export DLL functions: with the `__declspec(dllexport)` modifier or with a module-definition (`.def`) file. If you use the `__declspec(dllexport)` modifier, then it's not necessary to use a `.def` file. For more info, see [Exporting from a DLL](#).

# Creating an import library

An import library (`.lib`) file contains information that the linker needs in order to resolve external references to exported DLL functions, so that the system can locate the specified DLL and exported DLL functions at run-time. You can create an import library for your DLL when you build your DLL.

For more info, see [Building an import library and export file](#).

## Using an import library

For example, to call the `CreateWindow` function, you must link your code with the import library `User32.lib`. The reason is that `CreateWindow` resides in a system DLL named `User32.dll`, and `User32.lib` is the import library used to resolve the calls in your code to exported functions in `User32.dll`. The linker creates a table that contains the address of each function call. Calls to functions in a DLL will be fixed up when the DLL is loaded. While the system is initializing the process, it loads `User32.dll` because the process depends on exported functions in that DLL, and it updates the entries in the function address table. All calls to `CreateWindow` invoke the function exported from `User32.dll`.

For more info, see [Link an executable to a DLL](#).

## Related topics

- [Creating a Simple Dynamic-link Library](#)
- [DLLs \(Visual C++\)](#)
-

# Dynamic-link library search order

Article • 02/09/2023

It's common for multiple versions of the same dynamic-link library (DLL) to exist in different file system locations within an operating system (OS). You *can* control the specific location from which any given DLL is loaded by specifying a full path. But if you don't use that method, then the system searches for the DLL at load time as described in this topic. The *DLL loader* is the part of the operating system (OS) that loads DLLs and/or resolves references to DLLs.

## Tip

For definitions of *packaged* and *unpackaged* apps, see [Advantages and disadvantages of packaging your app](#).

## Factors that affect searching

Here are some special search factors that are discussed in this topic—you can consider them to be part of the DLL search order. Later sections in this topic list these factors in the appropriate search order for certain app types, together with other search locations. This section is just to introduce the concepts, and to give them names that we'll use to refer to them later in the topic.

- **DLL redirection.** For details, see [Dynamic-link library redirection](#).
- **API sets.** For details, see [Windows API sets](#).
- **Side-by-side (SxS) manifest redirection**—desktop apps only (not UWP apps). You can redirect by using an application manifest (also known as a side-by-side application manifest, or a fusion manifest). For details, see [Manifests](#).
- **Loaded-module list.** The system can check to see whether a DLL with the same module name is already loaded into memory (no matter which folder it was loaded from).
- **Known DLLs.** If the DLL is on the list of known DLLs for the version of Windows on which the application is running, then the system uses its copy of the known DLL (and the known DLL's dependent DLLs, if any). For a list of known DLLs on the current

system, see the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs`.

If a DLL has dependencies, then the system searches for the dependent DLLs as if they were loaded by using only their module names. That's true even if the first DLL was loaded by specifying a full path.

## Search order for packaged apps

When a packaged app loads a packaged module (specifically, a library module—a `.dll` file) by calling the [LoadPackagedLibrary](#) function, the DLL must be in the package dependency graph of the process. For more information, see [LoadPackagedLibrary](#). When a packaged app loads a module by other means, and doesn't specify a full path, the system searches for the DLL and its dependencies at load time as described in this section.

When the system searches for a module or its dependencies, it always uses the search order for packaged apps; even if a dependency is not packaged app code.

## Standard search order for packaged apps

The system searches in this order:

1. DLL redirection.
2. API sets.
3. **Desktop apps only (not UWP apps)**. SxS manifest redirection.
4. Loaded-module list.
5. Known DLLs.
6. The package dependency graph of the process. This is the application's package plus any dependencies specified as `<PackageDependency>` in the `<Dependencies>` section of the application's package manifest. Dependencies are searched in the order they appear in the manifest.
7. The folder the calling process was loaded from (the executable's folder).



8. The system folder (`%SystemRoot%\system32`).

If a DLL has dependencies, then the system searches for the dependent DLLs as if they were loaded with just their module names (even if the first DLL was loaded by specifying a full path).

## Alternate search order for packaged apps

If a module changes the standard search order by calling the [LoadLibraryEx](#) function with `LOAD_WITH_ALTERED_SEARCH_PATH`, then the search order is the same as the standard search order except that in step 7 the system searches the folder that the specified module was loaded from (the top-loading module's folder) instead of the executable's folder.

## Search order for unpackaged apps

When an unpackaged app loads a module and doesn't specify a full path, the system searches for the DLL at load time as described in this section.

### Important

If an attacker gains control of one of the directories that's searched, then it can place a malicious copy of the DLL in that folder. For ways to help prevent such attacks, see [Dynamic-link library security](#).

## Standard search order for unpackaged apps

The standard DLL search order used by the system depends on whether or not *safe DLL search mode* is enabled.

Safe DLL search mode (which is enabled by default) moves the user's current folder later in the search order. To disable safe DLL search mode, create the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode` registry value, and set it to 0. Calling the [SetDllDirectory](#) function effectively disables safe DLL search mode (while the specified folder is in the search path), and changes the search order as described in this topic.

If safe DLL search mode is enabled, then the search order is as follows:

1. DLL Redirection.
2. API sets.
3. SxS manifest redirection.
4. Loaded-module list.
5. Known DLLs.
6. **Windows 11, version 21H2 (10.0; Build 22000), and later.** The package dependency graph of the process. This is the application's package plus any dependencies specified as `<PackageDependency>` in the `<Dependencies>` section of the application's package manifest. Dependencies are searched in the order they appear in the manifest.
7. The folder from which the application loaded.
8. The system folder. Use the [GetSystemDirectory](#) function to retrieve the path of this folder.
9. The 16-bit system folder. There's no function that obtains the path of this folder, but it is searched.
10. The Windows folder. Use the [GetWindowsDirectory](#) function to get the path of this folder.
11. The current folder.
12. The directories that are listed in the `PATH` environment variable. This doesn't include the per-application path specified by the **App Paths** registry key. The **App Paths** key isn't used when computing the DLL search path.

If safe DLL search mode is *disabled*, then the search order is the same except that *the current folder* moves from position 11 to position 8 in the sequence (immediately after step 7. *The folder from which the application loaded*).

## Alternate search order for unpackaged apps

To change the standard search order used by the system, you can call the [LoadLibraryEx](#) function with **LOAD\_WITH\_ALTERED\_SEARCH\_PATH**. You can also change the standard search order by calling the [SetDllDirectory](#) function.

ⓘ **Note**

The standard search order of the process will also be affected by calling the [SetDllDirectory](#) function in the parent process before the start of the current process.

If you specify an alternate search strategy, then its behavior continues until all associated executable modules have been located. After the system starts processing DLL initialization routines, the system reverts to the standard search strategy.

The [LoadLibraryEx](#) function supports an alternate search order if the call specifies **LOAD\_WITH\_ALTERED\_SEARCH\_PATH**, and the *lpFileName* parameter specifies an absolute path.

- The standard search strategy begins (after the initial steps) in the calling application's folder.
- The alternate search strategy specified by [LoadLibraryEx](#) with **LOAD\_WITH\_ALTERED\_SEARCH\_PATH** begins (after the initial steps) in the folder of the executable module that [LoadLibraryEx](#) is loading.

That's the only way in which they differ.

If safe DLL search mode is enabled, then the alternate search order is as follows:

Steps 1-6 are the same as the standard search order.

7. The folder specified by *lpFileName*.
8. The system folder. Use the [GetSystemDirectory](#) function to retrieve the path of this folder.
9. The 16-bit system folder. There's no function that obtains the path of this folder, but it is searched.
10. The Windows folder. Use the [GetWindowsDirectory](#) function to get the path of this folder.
11. The current folder.

12. The directories that are listed in the `PATH` environment variable. This doesn't include the per-application path specified by the **App Paths** registry key. The **App Paths** key isn't used when computing the DLL search path.

If safe DLL search mode is *disabled*, then the alternate search order is the same except that *the current folder* moves from position 11 to position 8 in the sequence (immediately after step 7. *The folder specified by lpFileName*).

The [SetDllDirectory](#) function supports an alternate search order if the *lpPathName* parameter specifies a path. The alternate search order is as follows:

Steps 1-6 are the same as the standard search order.

7. The folder from which the application loaded.
8. The folder specified by the *lpPathName* parameter of [SetDllDirectory](#).
9. The system folder.
10. The 16-bit system folder.
11. The Windows folder.
12. The directories listed in the `PATH` environment variable.

If the *lpPathName* parameter is an empty string, then the call removes the current folder from the search order.

[SetDllDirectory](#) effectively disables safe DLL search mode while the specified folder is in the search path. To restore safe DLL search mode based on the **SafeDllSearchMode** registry value, and restore the current folder to the search order, call [SetDllDirectory](#) with *lpPathName* as NULL.

## Search order using `LOAD_LIBRARY_SEARCH` flags

You can specify a search order by using one or more `LOAD_LIBRARY_SEARCH` flags with the [LoadLibraryEx](#) function. You can also use `LOAD_LIBRARY_SEARCH` flags with the [SetDefaultDllDirectories](#) function to establish a DLL search order for a process. You can specify additional directories for the process DLL search order by using the [AddDllDirectory](#) or [SetDllDirectory](#) functions.

# Dynamic-Link Library Data

Article • 01/08/2021

A Dynamic-Link Library (DLL) can contain global data or local data.

## Variable Scope

Variables that are declared as global in a DLL source code file are treated as global variables by the compiler and linker, but each process that loads a given DLL gets its own copy of that DLL's global variables. The scope of static variables is limited to the block in which the static variables are declared. As a result, each process has its own instance of the DLL global and static variables by default.

### ⓘ Note

Your development tools may allow you to override the default behavior. For example, the Visual C++ compiler supports **#pragma section** and the linker supports the `/SECTION` option. For more information, see the documentation included with your development tools.

## Dynamic Memory Allocation

When a DLL allocates memory using any of the memory allocation functions ([GlobalAlloc](#), [LocalAlloc](#), [HeapAlloc](#), and [VirtualAlloc](#)), the memory is allocated in the virtual address space of the calling process and is accessible only to the threads of that process.

A DLL can use file mapping to allocate memory that can be shared among processes. For a general discussion of how to use file mapping to create named shared memory, see [File Mapping](#). For an example that uses the [DllMain](#) function to set up shared memory using file mapping, see [Using Shared Memory in a Dynamic-Link Library](#).

## Thread Local Storage

The thread local storage (TLS) functions enable a DLL to allocate an index for storing and retrieving a different value for each thread of a multithreaded process. For example, a spreadsheet application can create a new instance of the same thread each time the user opens a new spreadsheet. A DLL providing the functions for various spreadsheet operations can use TLS to save information about the current state of each spreadsheet (row, column, and so on). For a general discussion of thread local storage, see [Thread Local Storage](#). For an example that uses the [DllMain](#) function to set up thread local storage, see [Using Thread Local Storage in a Dynamic-Link Library](#).

**Windows Server 2003 and Windows XP:** The Visual C++ compiler supports a syntax that enables you to declare thread-local variables: `_declspec(thread)`. If you use this syntax in a DLL, you will not be able to load the DLL explicitly using [LoadLibrary](#) or [LoadLibraryEx](#) on versions of Windows prior to Windows Vista. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of `_declspec(thread)`.

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)

# Dynamic-link library redirection

Article • 10/13/2023

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*.

## DLL versioning problems

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.

## Loading and testing private binaries

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.

## Your options

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.

## Optional: configure the registry



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).

Console

```
reg add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options" /v DevOverrideEnable /t REG_DWORD /d 1
```

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

## Create a redirection file or folder

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.

## How to redirect DLLs for packaged apps

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`:

XML

```

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

XML

```

<ItemDefinitionGroup>
  <PreBuildEvent>

```

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

XML

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

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.

XML

```
if EXIST "<path-to-local-dll>" copy /y "<path-to-local-dll>"
$(IntDir)\microsoft.system.package.metadata\application.local 2>nul
if EXIST "<path-to-local-dll>" copy /y "<path-to-local-dll>"
$(IntDir)\microsoft.system.package.metadata\application.local 2>nul
</Command>
</PreBuildEvent>
```

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

XML

```
<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.

## Copying the binaries manually

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:

Console

```
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.

## Verify that everything worked

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.

## How to redirect DLLs for unpackaged apps

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).

---

## Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)

# Dynamic-Link Library Security

Article • 01/08/2021

When an application dynamically loads a dynamic-link library without specifying a fully qualified path name, Windows attempts to locate the DLL by searching a well-defined set of directories in a particular order, as described in [Dynamic-Link Library Search Order](#). If an attacker gains control of one of the directories on the DLL search path, it can place a malicious copy of the DLL in that directory. This is sometimes called a *DLL preloading attack* or a *binary planting attack*. If the system does not find a legitimate copy of the DLL before it searches the compromised directory, it loads the malicious DLL. If the application is running with administrator privileges, the attacker may succeed in local privilege elevation.

For example, suppose an application is designed to load a DLL from the user's current directory and fail gracefully if the DLL is not found. The application calls **LoadLibrary** with just the name of the DLL, which causes the system to search for the DLL. Assuming safe DLL search mode is enabled and the application is not using an alternate search order, the system searches directories in the following order:

1. The directory from which the application loaded.
2. The system directory.
3. The 16-bit system directory.
4. The Windows directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable.

Continuing the example, an attacker with knowledge of the application gains control of the current directory and places a malicious copy of the DLL in that directory. When the application issues the **LoadLibrary** call, the system searches for the DLL, finds the malicious copy of the DLL in the current directory, and loads it. The malicious copy of the DLL then runs within the application and gains the privileges of the user.

Developers can help safeguard their applications against DLL preloading attacks by following these guidelines:

- Wherever possible, specify a fully qualified path when using the [LoadLibrary](#), [LoadLibraryEx](#), [CreateProcess](#), or [ShellExecute](#) functions.
- Use the `LOAD_LIBRARY_SEARCH` flags with the [LoadLibraryEx](#) function, or use these flags with the [SetDefaultDllDirectories](#) function to establish a DLL search order for a process and then use the [AddDllDirectory](#) or [SetDllDirectory](#) functions to modify the list. For more information, see [Dynamic-Link Library Search Order](#).


**Windows 7, Windows Server 2008 R2, Windows Vista and Windows Server 2008:** These flags and functions are available on systems with [KB2533623](#) installed.

- On systems with [KB2533623](#) installed, use the `LOAD_LIBRARY_SEARCH` flags with the [LoadLibraryEx](#) function, or use these flags with the [SetDefaultDllDirectories](#) function to establish a DLL search order for a process and then use the [AddDllDirectory](#) or [SetDllDirectory](#) functions to modify the list. For more information, see [Dynamic-Link Library Search Order](#).
- Consider using [DLL redirection](#) or a [manifest](#) to ensure that your application uses the correct DLL.
- When using the standard search order, make sure that safe DLL search mode is enabled. This places the user's current directory later in the search order, increasing the chances that Windows will find a legitimate copy of the DLL before a malicious copy. Safe DLL search mode is enabled by default starting with Windows XP with Service Pack 2 (SP2) and is controlled by the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode` registry value. For more information, see [Dynamic-Link Library Search Order](#).
- Consider removing the current directory from the standard search path by calling [SetDllDirectory](#) with an empty string (""). This should be done once early in process initialization, not before and after calls to [LoadLibrary](#). Be aware that [SetDllDirectory](#) affects the entire process and that multiple threads calling [SetDllDirectory](#) with different values can cause undefined behavior. If your application loads third-party DLLs, test carefully to identify any incompatibilities.
- Do not use the [SearchPath](#) function to retrieve a path to a DLL for a subsequent [LoadLibrary](#) call unless safe process search mode is enabled. When safe process search mode is not enabled, the [SearchPath](#) function uses a different search order than [LoadLibrary](#) and is likely to first search the user's current directory for the specified DLL. To enable safe process search mode for



the **SearchPath** function, use the [SetSearchPathMode](#) function with `BASE_SEARCH_PATH_ENABLE_SAFE_SEARCHMODE`. This moves the current directory to the end of the **SearchPath** search list for the life of the process. Note that the current directory is not removed from the search path, so if the system does not find a legitimate copy of the DLL before it reaches the current directory, the application is still vulnerable. As with [SetDllDirectory](#), calling **SetSearchPathMode** should be done early in process initialization and it affects the entire process. If your application loads third-party DLLs, test carefully to identify any incompatibilities.

- Do not make assumptions about the operating system version based on a [LoadLibrary](#) call that searches for a DLL. If the application is running in an environment where the DLL is legitimately not present but a malicious copy of the DLL is in the search path, the malicious copy of the DLL may be loaded. Instead, use the recommended techniques described in [Getting the System Version](#).

The Process Monitor tool can be used to help identify DLL load operations that might be vulnerable. The Process Monitor tool can be downloaded from <https://technet.microsoft.com/sysinternals/bb896645.aspx> .

The following procedure describes how to use Process Monitor to examine DLL load operations in your application.

#### To use Process Monitor to examine DLL load operations in your application

1. Start Process Monitor.
2. In Process Monitor, include the following filters:
  - Operation is CreateFile
  - Operation is LoadImage
  - Path contains .cpl
  - Path contains .dll
  - Path contains .drv
  - Path contains .exe
  - Path contains .ocx
  - Path contains .scr

- Path contains .sys

3. Exclude the following filters:

- Process Name is procmon.exe
- Process Name is Procmon64.exe
- Process Name is System
- Operation begins with IRP\_MJ\_
- Operation begins with FASTIO\_
- Result is SUCCESS
- Path ends with pagefile.sys

4. Attempt to start your application with the current directory set to a specific directory. For example, double-click a file with an extension whose file handler is assigned to your application.

5. Check Process Monitor output for paths that look suspicious, such as a call to the current directory to load a DLL. This kind of call might indicate a vulnerability in your application.

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)

# Applnit DLLs and Secure Boot

Article • 09/30/2022


Starting in Windows 8, the Applnit\_DLLs infrastructure is disabled when secure boot is enabled.

## About Applnit\_DLLs

The Applnit\_DLLs infrastructure provides an easy way to hook system APIs by allowing custom DLLs to be loaded into the address space of every interactive application. Applications and malicious software both use Applnit DLLs for the same basic reason, which is to hook APIs; after the custom DLL is loaded, it can hook a well-known system API and implement alternate functionality. Only a small set of modern legitimate applications use this mechanism to load DLLs, while a large set of malware use this mechanism to compromise systems. Even legitimate Applnit\_DLLs can unintentionally cause system deadlocks and performance problems, therefore usage of Applnit\_DLLs is not recommended.

## Applnit\_DLLs and secure boot

Windows 8 adopted UEFI and secure boot to improve the overall system integrity and to provide strong protection against sophisticated threats. When secure boot is enabled, the Applnit\_DLLs mechanism is disabled as part of a no-compromise approach to protect customers against malware and threats.

Please note that secure boot is a UEFI protocol and not a Windows 8 feature. More info on UEFI and the secure boot protocol specification can be found at <https://www.uefi.org> .

# Applnit\_DLLs certification requirement for Windows 8 desktop apps

One of the certification requirements for Windows 8 desktop apps is that the app must not load arbitrary DLLs to intercept Win32 API calls using the Applnit\_DLLs mechanism. For more detailed information about the certification requirements, refer to section 1.1 of [Certification requirements for Windows 8 desktop apps](#).

## Summary

- The Applnit\_DLLs mechanism is not a recommended approach for legitimate applications because it can lead to system deadlocks and performance problems.
- The Applnit\_DLLs mechanism is disabled by default when secure boot is enabled.
- Using Applnit\_DLLs in a Windows 8 desktop app is a Windows desktop app certification failure.

To download a whitepaper with info about Applnit\_DLLs on Windows 7 and Windows Server 2008 R2, visit the [Windows Hardware Dev Center Archive](#), and search for *Applnit DLLs in Windows 7 and Windows Server 2008 R2*.

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)