Installed Location Virtualization

An under-documented new MSIX feature

that we need!

A Community White Paper by:

Tim Mangan

TMurgent Technologies, LLP

Introduction

We are constantly looking for ways to improve the compatibility of running traditional Win32 and Dot Net Framework based applications inside an MSIX container. This often includes adding the Package Support Framework to the MSIX package, but there are other lesser-understood features that were not originally part of MSIX but were added without fanfare or even much documentation.

The under-documentation of these features is in part due to how they get added. Generally we see the definition of how to specify the feature in the AppXManifest by scouring MSIX extension schemas as they get released. But these schema extensions only provide syntax but don't tell us what they do or why you might want to use the feature. They also appear before implementations are added into the client operating system so we can't just try to turn them on either as they are ignored.

Eventually, the feature support is added to a new OS version, usually a semi-annual release but could appear in a monthly feature release, but to date Microsoft doesn't seem to add it into the release notes or even blog about it.

InstallationLocationVirtualization (ILV) is one such feature. I detected in in the UAP10 schema extensions many years ago, and based on the name along I felt it might be important, although the **PackageWriteRedirectionCompatibilityShim** sounds somewhat similar and is also under-documented in the Desktop7 schema extensions which appeared earlier. Other testing has shown that this feature only affects file requests made to the primary folders of the package application path (not VFS paths). This significant restriction makes it a poor candidate for most repackaging situations.

My requests for documentation on these features have not produced much response, although now the item syntax now appears in the official online documentation for each. For *InstallationLocationVirtualization,* the only current documentation is this one page here https://learn.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap10-installedlocationvirtualization, which includes a single paragraph description:

> This extension is intended to be used by desktop apps in an MSIX package that write to their installation directory. These types of apps cannot normally write to their installation directory, so this extension redirects the write operations to a safe location in the app data. This extension also enables you to specify what happens during app updates to files in the app's installation directory that were previously modified, added, or deleted by the app.

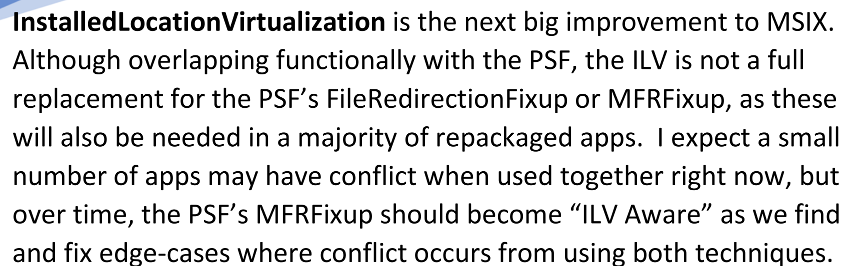
The documentation does not identify what OS version is required to be able to use this feature, but based on other UAP10 features where Microsoft has specified a minimum version, it should be either 21H1 or 21H2. I have verified that the feature works on both Windows 10 and Windows 11 21H2. If you are running older versions of the OS in production you should verify the feature availability yourself.

So what is *InstallationLocationVirtualization* and why should you be interested? This paper attempts to answer this. The information in this paper is not a substitute for Microsoft Documentation that we hope will eventually appear. It is the result of some detailed testing, analysis using standard and home-made tools in a research that was not terribly unlike reverse-engineering.

Summary for those afflicted with a short attention span

In short, ILV is a package-level extension that may be added to the AppXManifest file to aid in application compatibility, especially when package files must be altered. The OS support for this extension was added at an undocumented date, but probably 21H1, so we are nearly at a point that we can assume support in all production OSs.

The ILV is implemented mostly in the bindflt and wcifs mini-filter drivers in the kernel, but there also appears to be some support in what I call the "MSIX runtime" software at the user-kernel mode boundary. By leveraging filter drivers, this provides for improved support over the previous *PackageWriteRedirectionCompatibilityShim*.



> **InstalledLocationVirtualization** is the next big improvement to MSIX. Although overlapping functionally with the PSF, the ILV is not a full replacement for the PSF's FileRedirectionFixup or MFRFixup, as these will also be needed in a majority of repackaged apps. I expect a small number of apps may have conflict when used together right now, but over time, the PSF's MFRFixup should become "ILV Aware" as we find and fix edge-cases where conflict occurs from using both techniques.
>
> Tim Mangan

The feature is appears to provide the ability for an application to write to, or delete, any file path within the package root folder and below (including VFS paths). This is implemented by redirection to a per-user shadow folder for the package, the same %localappdata%\packages\packagefamilyname\localcache area used by the PSF. (Note that this is LocalAppData and not AppData as mentioned in the existing limited documentation).

This support is extended to requests made to the shadow area back to see package files. Applications that use native path requests will also be affected, but only if the MSIX runtime or PSF first provide a mapping. So if you needed the PSF before for AppData, LocalAppData, and AppVPackageDrive files, for example, you'll still need the PSF.

Specifically, this feature is known to be needed for applications that write their configuration settings using the System.Configuration API as the PSF is unable to support that API. Without the ILV support those apps will at a minimum be unable to save settings, but I have seen apps crash or issue error dialogs to reinstall. Apps that use other APIs to write to package files may use either the ILV or PSF to solve the write problem.

The use of ILV will change some failure return codes to the app, specifically returning some PathNotFound errors rather than the AccessDenied message that the app would have seen natively. This is something likely to cause a problem in some apps when the feature starts getting used without additional support.

The compatibility with both ILV with the PSF FileRedirectionFixup or MFRFixup is under investigation, however looks promising. It certainly avoids the issue mentioned in the previous paragraph as file paths

would get redirected for those cases before the ILV kicks in. In all likelihood, I will find that I need to add an "ILV aware" setting to the MFRFixup for best compatibility. It probably makes no sense to have the MFRFixup do things that the ILV will do for us, and such a mode could improve performance and possibly avoid edge-case issues. The MFRFixup is nearing its initial release and that release will not have the ILV-aware support added.

The following table shows the success or failure of certain simple operations that are made by an application using the specified path when the file is present in the package. Green represents success, red is failure.

R=Open file for read
W=Open file for write
D=Delete file

File Path Requested by app	MSIX	MSIX+ILV	MSIX_MFR	MSIX+ILV+MFR
C:\Program Files\Vendor	R W D	R W D	R W D[1]	R W D
<Root>\VFS\ProgramFilesX64\Vendor	R W D	R W D	R W D[1]	R W D
%localappdata%\Vendor	R W D	R W D	R W D[1]	R W D
<Root>\VFS\Local AppData\Vendor	R W D	R W D	R W D[1]	R W D
Modern "System.Configuration"[2]	R W	R W	R W	R W

The remainder of this document covers the background around this feature, details that have been discovered about the implementation, examples and information on some of the tooling used to investigate.

[1] The MFR will fail to delete the file, but return a successful response for app compatibility.
[2] App that uses System.Configuration API to store configuration information. Failure to write often leads to app crash or reinstall prompts.

Background

MSIX applications run inside a specially configured Windows Container that provide many file based benefits. One of those benefits is that all files in the package live in a special namespace that defaults to the 'C:\Program Files\WindowsApps\{FullPackageName}' folder, and those files are protected from modification after installation.

Furthermore, there are a set of folders for which the application may request a file from where the app normally would have found the file rather than from within the package folder namespace. So, for example, a request for C:\Windows\System32\xxx.dll on an x64 OS system would result in a check to see if it exists under 'C:\Program Files\WindowsApps\{FullPackageName}\VFS\SYSTEMX64\xxx.dll' first, and if not look in the native location. While there are many VFS folders that may be defined in the package namespace, the MSIX runtime only provides this mapping for a subset, as documented in the *Packaged VFS Locations* heading of this document: [Understanding how packaged desktop apps run on Windows - MSIX | Microsoft Learn (https://learn.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the-scenes)](https://learn.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the-scenes).

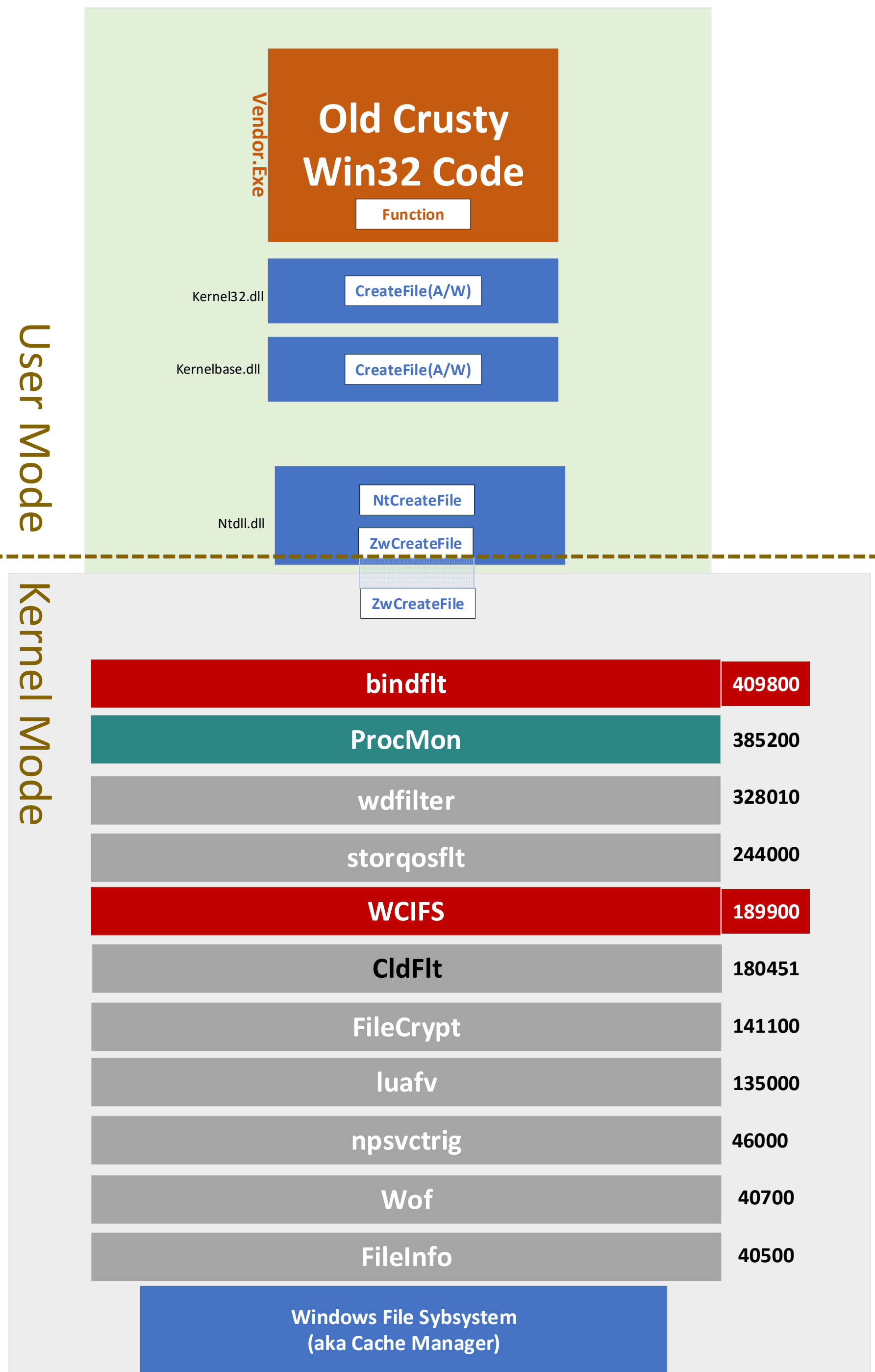
For the application to have visibility to other VFS files, especially those in the package under VFS\AppData, VFS\LocalAppData, and VFS\AppVPackageDrive, the Package Support Frameworks *FileRedirectionFixup* was used. This fixup not only adds in the missing mappings from native path requests to the package, it also supports a redirection of file requests by making a copy of the package file to an alternate area under '%localappdata%\package\{packagename}\LocalCache\Microsoft\VFS\WritablePackageRoot', making is possible for the app to update files. Normally this is configured to prevent executable files from being updated for security reasons.

MSIX Runtime and Mini-Filter Drivers

The ***MSIX Runtime*** is a term that I use for software that implements must of this VFS redirection. This software appears to live at the User/Kernel boundary, likely inside files like NtCreateFile and/or ZwCreateFile.

In addition to the MSIX Runtime, there is also MSIX support provided by the ***binflt*** [3]and ***wcifs*** mini-filter drivers located below this in the kernel of the OS. This can be depicted as a drawing, shown next, which I use in my training classes. In the drawing you see the application at the top making a call to open a file using the Windows API CreateFile. This in turn calls NTCreate File and then ZwCreateFile which is involved in the transition from user mode to kernel mode. The call then passes through many mini-filter drivers (which may or may not have support for this call) and finally into the file system implementation in the Windows Cache Manager and potentially out to the disk.

[3] Bindflt (which runs at Altitude 409800) appears to be a replacement of wcnfs (409900) mini-filter driver that was is prior versions of Windows 10. It is possible that this coincides with ILV support being available.



Bindflt is believed to provide "namespace support" and appears to replace the older wcnfs mini-filter driver which provided container file system support. The wcifs driver is thought of as a holdover from the server container implementation for protocol virtualization (IP Address Spoofing) that is not needed

in the desktop container, however there is file based functionality implemented as part of ILV in this driver.

PSF FileRedirectionFixup and MFRFixup

The *FileRedirectionFixup* is currently the most used technique for providing a variety of file based compatibility fixes for legacy apps under MSIX. It is often considered complicated to configure and overly ambitious with making file copies as it uses a copy-on-access algorithm.

A new fixup, the *MFRFixup* is just now becoming available as a potential replacement for the *FileRedirectionFixup.* This replacement is re-architected to solve many of the challenges of the FileRedirectionFixup, including support for copy-on-write, reverse redirection from the redirection area, and redirection to local for things like the Documents folder where we don't want files trapped in the app's private area.

These fixups are implemented in user mode by intercepting application calls into WindowsAPI functions like CreateFile.

These fixups are mostly used in situations where traditional Win32 and .Net Framework applications are repackaged into MSIX, however developers can and do sometimes use them to release older code as MSIX as well.

PackageWriteRedirectionCompatibilityShim

Microsoft added this feature to MSIX prior to the ILV. Like the ILV, this feature is similarly under-documented (see [App capability declarations - UWP applications | Microsoft Learn (https://learn.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations)](https://learn.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations) and I have not spent time to understand the implementation. The feature is triggered by requesting this as a Capability in the AppXManifest file.

The purpose of this shim seems to be somewhat similar in aims to that of the ILV, that is it is designed to allow applications to write to package files. Unfortunately the feature required the app to request the actual package path. I think that this is because it is implemented above the MSIX Runtime, and may be using Detours to intercept at about the same level as the PSF fixups, just built into the OS.

The limitation regarding VFS redirection has limited the usefulness of this feature, but it can be helpful in certain smaller applications.

Modifiable App

A different AppXPackage capability for a seemingly similar purpose, this seems to have been intended for either UWP apps, or as a form of private fix for a game development company. I have never used it.

InstalledLocationVirtualization

The ILV is triggered by a package level extension added to the AppXManifest File. An example of this syntax is shown :

```xml
<Extensions>

        <uap10:Extension
          Category="windows.installedLocationVirtualization">
            <uap10:InstalledLocationVirtualization>
            <uap10:UpdateActions
              ModifiedItems="keep"
              DeletedItems="reset"
              AddedItems="keep"/>
            </uap10:InstalledLocationVirtualization>
        </uap10:Extension>
    </Extensions>
```

The “UpdateActions” settings appear to only affect what happens when the package is later upgraded. The normal behavior without ILV is to keep all types of changes that are stored in the redirection area when a package is upgraded, so I recommend using keep for all three action types. I believe the action types have the following meaning:

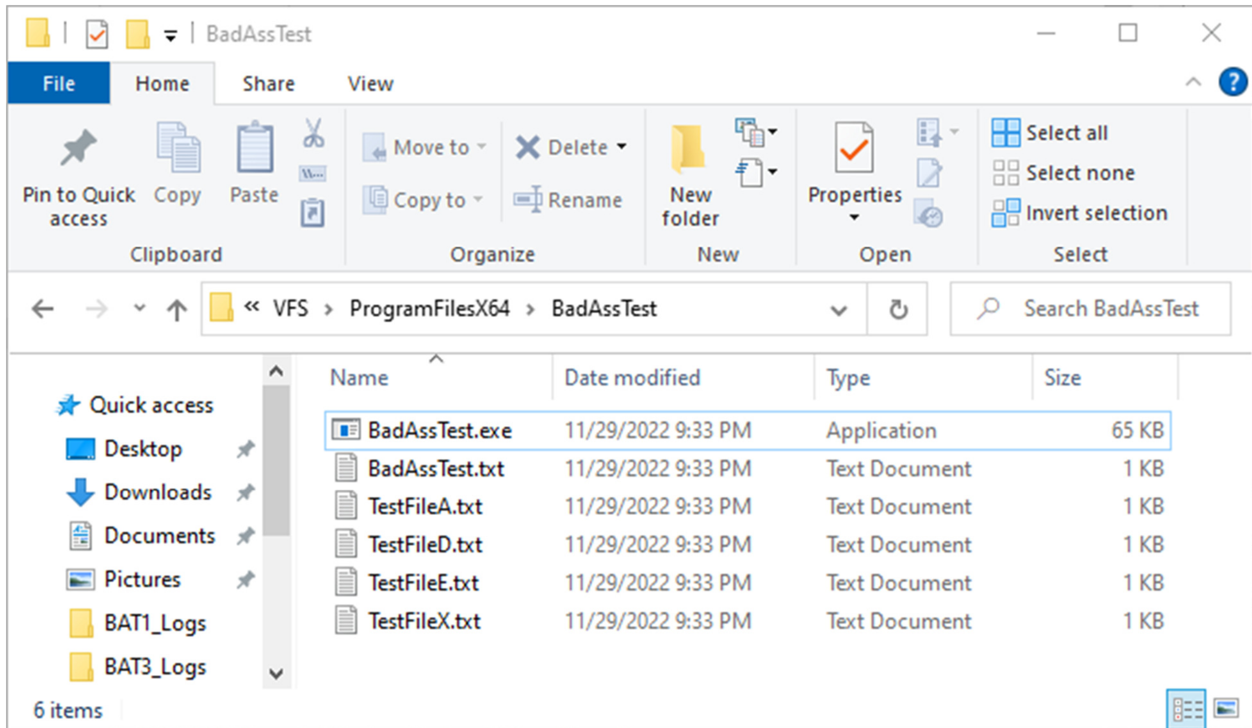
- **ModifiedItems** are modified copies of existing package files.
- **AddedItems** are new files that were added by the application (the same path/name does not appear in the package folder) and ended up in the redirection area because the app tried to write it into the package folder or directly to the redirection area.
- **DeletedItems** are deletion markers added to the redirection area that hide the presence of the file, even when it is in the package. Deletion markers require ILV for support so this is something new, which is covered in the Details section of this document.

It is interesting that Microsoft chose to configure the update scenario in such detail in the AppXManifest rather than make it a choice in powershell as part of something similar to the repair function.

Examples

I used a home-grown test application as part of my investigation, '*BadAssTest*'. Information on this tool is in the Tools section at the end of of this paper. This is a Win32 Console application that will make certain filesystem calls to certain paths.

The image below shows package files for the installation folder.

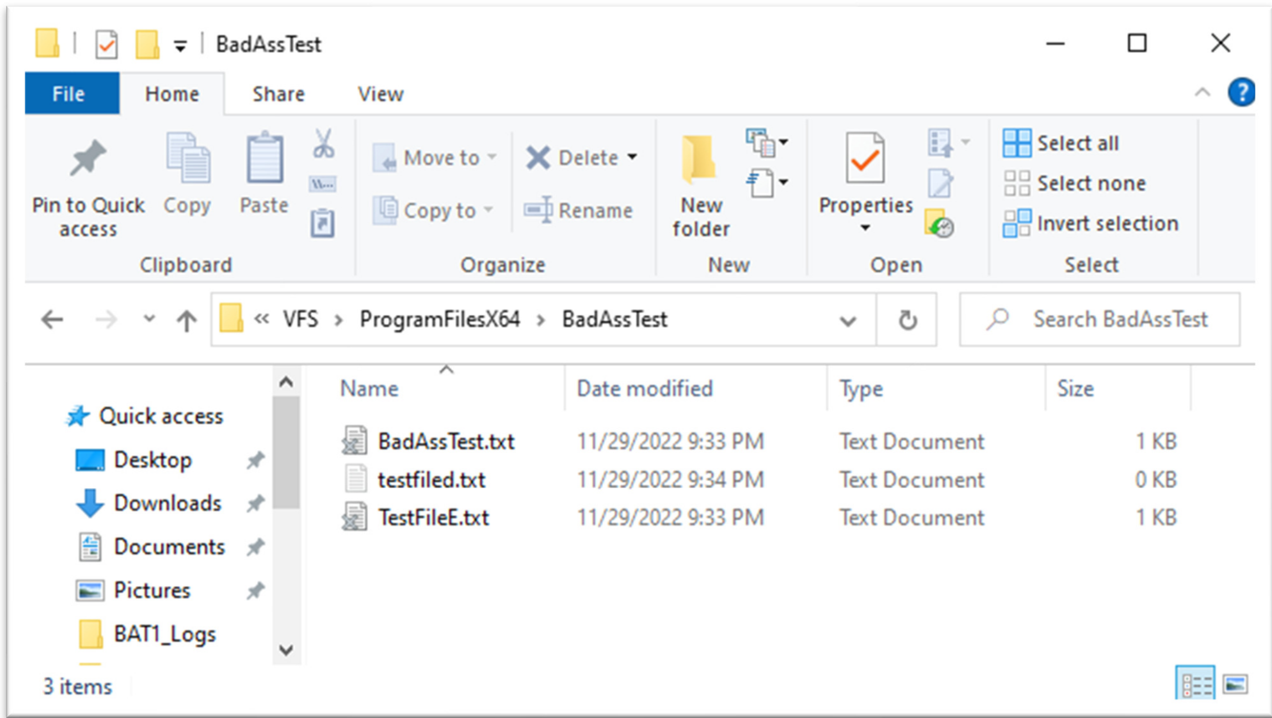


The BadAssTest.txt file is the configuration file for the exe and the other files are used in file access testing. The contents of this file are shown here:

```
C:\Program Files\BadAssTest\TestFileA.txt,1,2,3,4,5
C:\Users\Admin\AppData\Local\BadAssTest\TestFileB.txt,1,2,3,4,5
C:\BadAssTest\TestFileC.txt,1,2,3,4,5
<Root>\VFS\ProgramFilesX64\BadAssTest\TestFileD.txt,1,2,3,4,5
<WritablePackageRoot>\VFS\ProgramFilesX64\BadAssTest\TestFileE.txt,1
<Root>\VFS\ProgramFilesX64\BadAssTest\TestFileE.txt,1
<WritablePackageRoot>\VFS\ProgramFilesX64\BadAssTest\TestFileE.txt,1
<WritablePackageRoot>\VFS\ProgramFilesX64\BadAssTest,6
<Root>\VFS\ProgramFilesX64\BadAssTest,6
```

I will only be showing results from the lines associated with the application folder, but the other lines help to fill out the picture of what ILV does.

When using the PSF FileRedirectionFixup or MFRFixup, when you look at the redirection folder you'll see normal looking files. But with the ILV in place, the display looks very similar to that of the Microsoft App-V cache:

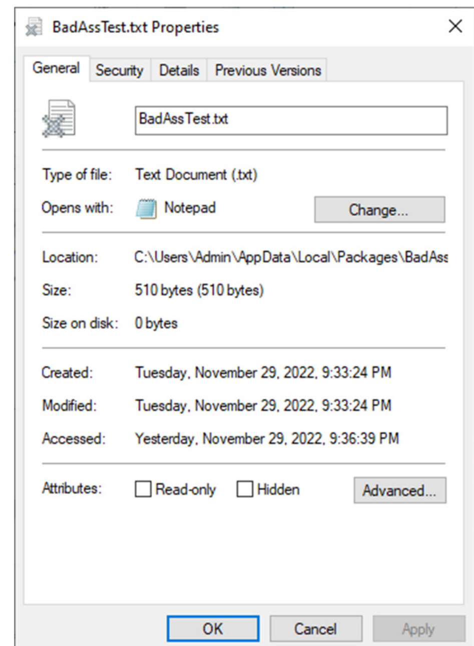


The grey X over the icons seems to occur when the file is marked with the SparseFile marking in the extended attributes.

Our experience with Microsoft App-V is that this meant that less than 100% of the actual file was present on disk. Such files would start out with an empty file (well the EA takes of 4k so there is that) and that if you ask for the file properties you'd see a difference between the full file size and the size on disk. App-V uses this information to determine when additional file blocks need to be streamed over to this system by the AppVStream mini-filter driver.

With ILV, it seems that the wfsflt mini-system driver uses it siimilarly, although file contents are really an all or nothing thing, so if any file contents are to be updated, the full file is written here(copied and modified).

Here is a view of the properties of that first file. Notice the difference on Size and Size on disk fields.



My TestDirectory program shows more detail about sparse file, although it is more useful in the App-V case as it also shows what parts of the file are present on disk. (See section at the end of this document for more information on this tool).

Another thing to notice in the file list of the redirected area is that it did not include all files that were present in the package folder. With App-V, the packaging publish process would pre-create all of the placeholder files up front. In a large package this would significantly slow down this process making it take seconds to minutes for the app to be ready. Even with ILV, the MSIX AppInstaller does not create these placeholders. Instead, they are created whenever the package file is accessed. This matches our test cases for std::filesystem::exists, Get/SetFileAttributes, and CreateFile. Furthermore, if you specifically make one of those calls against the filesystem path in the redirection area, wcifs will also trigger the creation of the placeholder. This also occurs if the application requested the native file path, but another component (either PSF or the MSIX Runtime) redirected this into the package or redirection equivalent.

Finally, in the file list of the redirected area, we normally would not have seen the file entry 'testfiled.txt'. It was necessary to change the settings in explorer to show hidden and system files for this entry to show. That is because test number 5 was run against the TestFileD.txt file of the program folder. Without the ILV this call would have failed, but with the ILV the system puts a placeholder file in the redirection folder (changing the name to lowercase for some reason) but it has no contents (both file size and size on disk are 0), is not marked sparse, and is set with hidden and system attributes.

Attempts to test the file for existence of the file or read once deleted (in the package or redirected pathing) will result in a FileNotFound return.

Applications will look for files by either Opening and querying a directory, or using something like the FindFirstFile/FindNext file API. Ultimately a directory contains the list of files under it. The FileRedirectionFixup and MfrFixup have extensive logic to layer local, package, and redirected folders into a single list that it provides the apps. From the testing shown here, it appears that the ILV has similar logic built into the mini-filter driver. So unlike the partial list of files we see in the explorer when looking at the redirection area, the test calling FindXXFile shows the overlaid view of the redirection area on top of the package files. Similarly, a FindXXFile call on the program folder will show the same overlay. This includes new files that only exist in the redirected area.

If an application queries an application directly, it opens a directory handle and then queries that handle for the files. With user mode provided redirections (such as used in the PSF) this means that the app will only see the files in whichever copy of the directory was actually opened. This is the most common source of remaining compatibility issues for MSIX. Unlike a user-mode component, a filter driver has the possibility of solving this. This is commonly done in App-V and other layering solutions, however I have not yet determined if or how this works with ILV.

Details

The findings of my investigation are as follows:

- *InstallationLocationVirtualization* is a package-level extension that may be added to the AppXManifest that appears to impact the BindFlt and WCIFS mini-filter drivers inside the windows kernel , but also possibly the MSIX runtime software near the User-Kernel mode boundary (but before any mini-filter drivers).
- With the feature enabled, application files inside the package may have a virtualized copy added to the package redirection area where they may be written to.
- Read-only requests to certain files can cause empty file entries added to the redirection area using sparse markings in the extended attributes not unlike App-V cache files or OneDrive offline files. Supported requests include:
    - Those at the root of the package and non-VFS subfolders.
    - Those under *any* package VFS folder.
    - Those requested using native paths that the MSIX runtime automatically checks the equivalent VFS folder. Note that package files in VFS folders like AppData, LocalAppData, and AppVPackageRoot are not subject to this initial redirection.
- New files added to the package folders (both root and VFS folders) will be added to the redirection area.
- Requests to write to certain files will cause a copy to be made in the redirection area and the file will be opened from there. Support is for:
    - Those at the root of the package and non-VFS subfolders.
    - Those under *any* package VFS folder.
    - Note that native path requests, including those that map for read-only purpose do not map for this purpose, and the app may receive a PathNotFound error instead of a AccessDenied error which might confuse the app.
- The ILV also supports the deletion of a package file, although done via trickery.
    - If an attempt is made to delete the package file, a zero-byte file is created in the redirection area, using a lower-case version of the filename, and it is marked as both Hidden and System.
    - This is used as a trigger for later accesses to not find the file even if in the package.
    - The deleted file does not show up in a list of the files of either the package or redirection folder in FindFirst/Next File.
- When ILV is in use:
    - Explorer will show directory listing of the redirection area with only the list files that have been accessed (including existence checks). This access could have occurred in a request by either folder, or a read request from a mapped VFS native path. Deleted files are not shown by default, but may be seen by viewing hidden system files.
    - Explorer will show directory listing of the package are with all package files, even if there is a deletion marker for it in the deleted area.
    - FindFirstFile on the package folder will show all package files, except for any with a deletion marker in the folder.
    - FindFirstFile on the redirection folder will show all package files, except for any with a deletion marker in the redirection folder, including those not yet accessed.

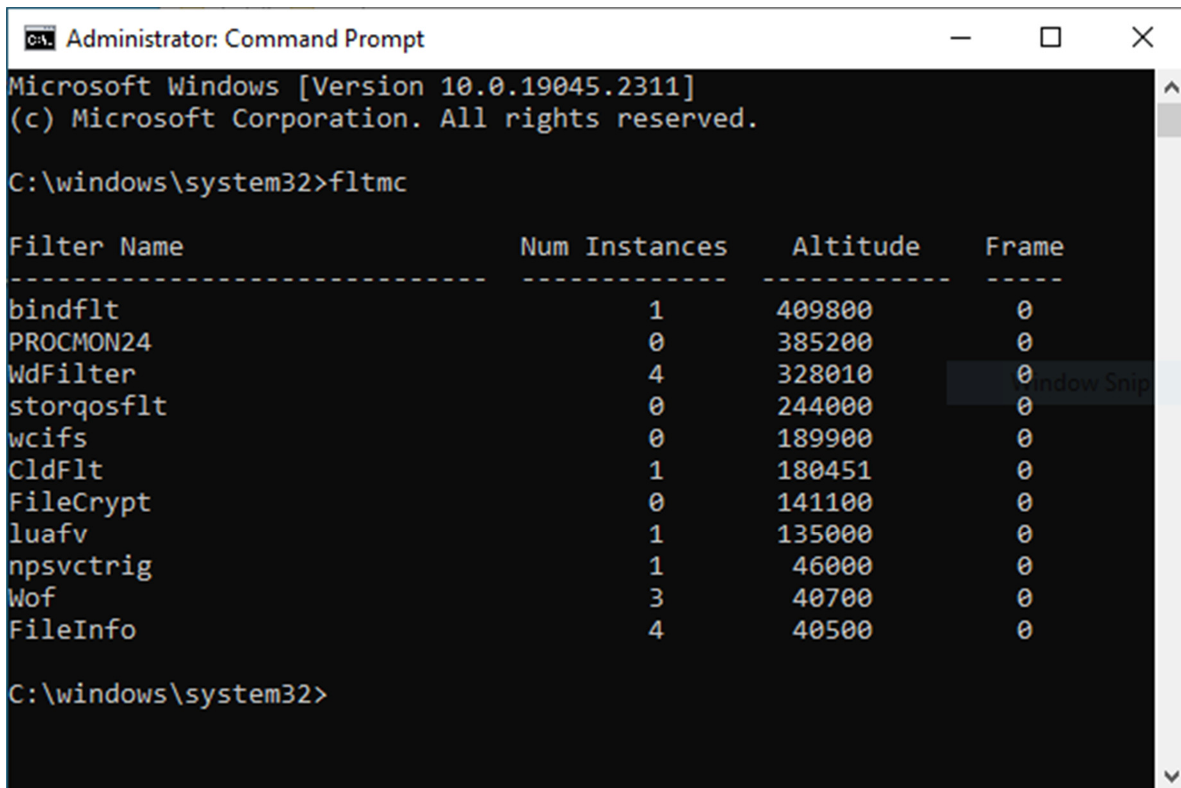
    - Should the app try to access a file (exists, Attributes, Open for read or write) in either the package or redirection folder that is not currently visible in the redirection area in explorer (other than deletion marker), this will also trigger the placeholder which will then be accessed. FindFirstFile does not trigger this.
    - Note that OLV is not a true overlay layering of this folder over the package folder, but close.
- ILV is implemented quite differently than the PSF *FileRedirectionFixup* and *ManagedFileFixup*, as well as the *PackageWriteRedirectionCompativilityShim* which are user-mode intercepts. Implemented as a filter driver, the ILV has impact to file calls that avoid these components. One known case where this makes a difference is DotNet Framework apps that use System.Configuration API to read and store application settings. These applications tend to crash or fail when saving application settings without the ILV.
- The ILV can replace the need of the PSF FileRedirectionFixup or ManagedFileFixup in some cases, but more often it be needed to augment these compatibility fixes in other cases. The ILV by itself does not make the non-mapped VFS pathing visible (for example if the app looks to AppData for a configuration file).
- Experience in insufficient at this time to determine if ILV might cause issues to some packages, but at this time I recommend adding the feature by default and try removing it if the package has a problem.
- ILV does not provide protection against package binary files being modified. We consider it a best practice to prevent this in the FileRedirectionFixup/MfrFixup. Combining these currently allows for the overwrite, something that might get addressed in an ILV-aware setting for the MfrFixup in the future.
- The test results showing certain file requests being handled by bindfs without passing through lower altitude filter drivers may be of concern, and testing with more unusual filter drivers may be warranted. Here, I am especially concerned with drivers that perform their own layering or user profile operations, such as the Microsoft FsLogix, Ivanti, Citrix UPD, and VMWare Volumes, and others.

Tools and Techniques used in creation of this paper

A number of tools were used in the creation of this paper. Those wishing to perform their own investigations might find them handy.

Fltmgr.exe

Fltmgr is a built in OS tool to discover what mini-filter drivers are currently present in memory on your system. You use a command prompt that has been elevated to be able to run the tool, but just typing FltMgr should give you an ordered list by altitude such as is shown below:



Fltmgr has additional capabilities to detail the instances of a mini-filter driver. The instance is basically the namespace the driver is interested in. Mostly this mean a volume like C:, but if you looked at npsvctrig's instance you'd quickly realize that it is involved only with named pipes.

Fltmgr also has the ability to unload an instance, and to load a new instance at a specified altitude, however I have found that this is unreliable so I use the registry technique described for Process Monitor.

Process Monitor

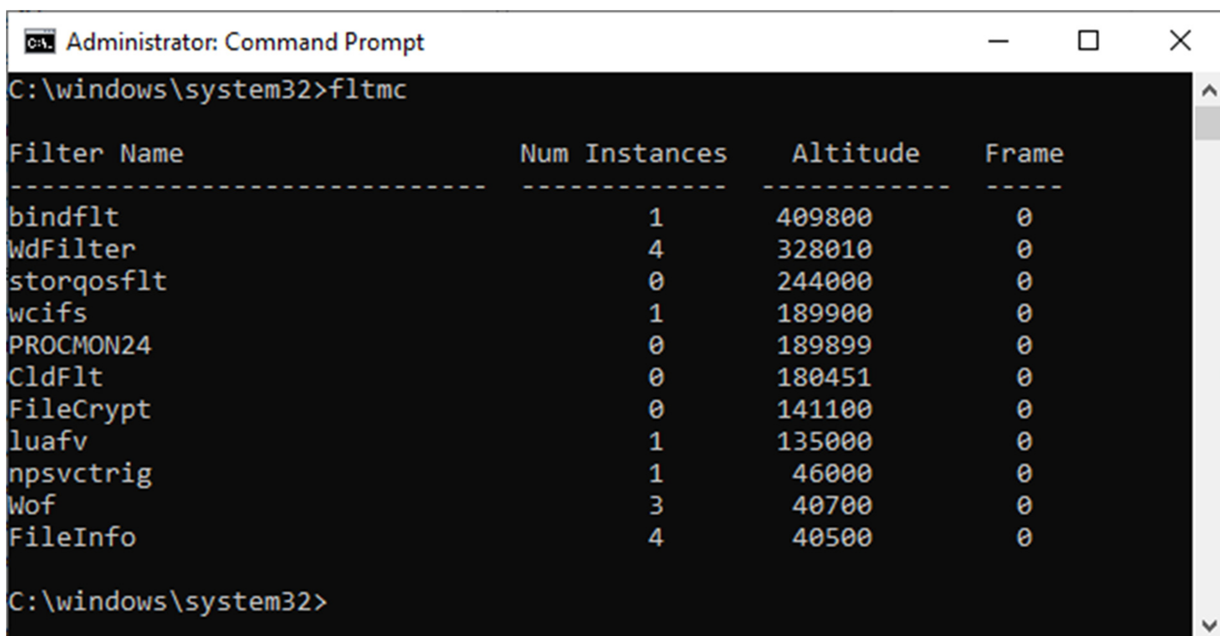
Process Monitor is a great tool to view file (and registry) activity. The capture portion is implemented as a mini-filter driver with an altitude of 385200, which is just below the bindflt mini-filter driver. This means that (by default), Process monitor will show activity after the *PSF*, *PackageWriteCompatibilityShim*, *MSIX Runtime,* and *bindflt* have potentially made changes to the call. Often these changes may be to try a different, or multiple possible filepath or even change parameters.

Generally, but not always this leads to more ProcMon captured events than what the application requested. But I have seen that there are times where something like bindflt might eat the request and respond without passing the request down to the actual file system! This has lead me to believe that binflt holds in memory information about the existence of package files by name/path (or has some other out of band means to view Cache Manager details that avoids lower mini-filter drivers to determine this).

It is possible to move the Process Monitor filter location by changing the altitude used. I discovered this through trial and error over 15 years ago and use the technique for debugging and examining mini-filter driver behavior all the time. The technique that I use, which is documented in many places on the web today, is to:

- Alter the registry value 'Altitude' under the key 'HKLM\SYSTEM\CurrentControlSet\Services\PROCMON24\Instances\Process Monitor 24 Instance'.
- The number 24 seems to change only when Microsoft updates this driver, which is much less frequent than changes to the tool, but at some point that key will change because the driver changed. If you don't see any PROCMON## key then you need to start Process Monitor once to get values in place.
- Choose the altitude that you want Process Monitor to use. It needs to be unique, so put it just above or below the driver you are interested in.
- After changing the value to 'Altitude' to a new number, you must modify the permissions on the 'Process Monitor 24 Instance' key. You must then reboot so that the driver can be reloaded at the new altitude when Process Monitor starts capturing.

After doing this, always verify the new altitude using fltmc. The image below shows the driver moved just below the wcifs driver.



```
Administrator: Command Prompt
C:\windows\system32>fltmc

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
bindflt                                 1       409800         0
WdFilter                                4       328010         0
storqosflt                              0       244000         0
wcifs                                   1       189900         0
PROCMON24                               0       189899         0
CldFlt                                  0       180451         0
FileCrypt                               0       141100         0
luafv                                   1       135000         0
npsvctrig                               1        46000         0
Wof                                     3        40700         0
FileInfo                                4        40500         0

C:\windows\system32>
```

When doing this kind of testing, I prefer to have a very simple application scenario where I know exactly what the app is going to request. This may be due to other forms of monitoring, or having access to the code. I typically will write a simple app to perform just that one task, then run traces with the ProcMon driver loaded at different altitudes and compare the results. This provides great evidence of what goes in the black box and what comes out the other side.

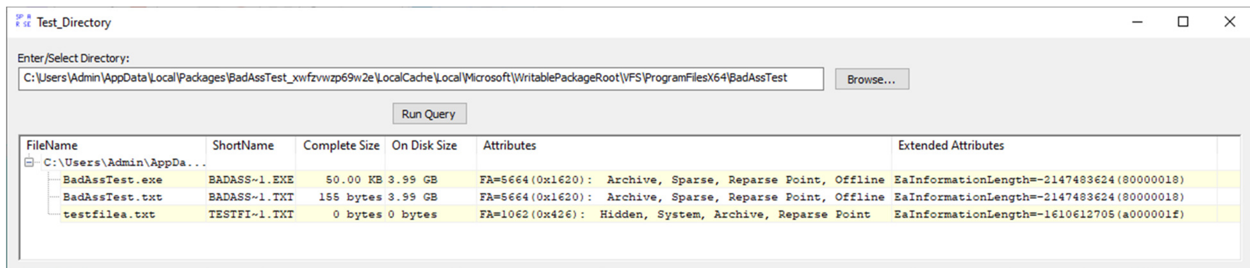
Additional discussion of Process Monitor is in the Advanced Process monitor section later in this paper.

TestDirectory.exe

TestDirectory is a free tool I wrote to look at the effect of the filter drivers in Microsoft App-V. See [Test_Directory: Tool for investigating App-V Cache Completeness (tmurgent.com) (https://www.tmurgent.com/APPV/en/resources/tools-downloads/tools-downloads-appv5/89-tools/app-v-5-tools/166-test-directory)](https://www.tmurgent.com/APPV/en/resources/tools-downloads/tools-downloads-appv5/89-tools/app-v-5-tools/166-test-directory).

The tool also proved invaluable to my ILV investigation as the tool provides information about the extended attributes and reparse tagging used by pretty much any files supported by a layering filter driver.

Here is an example when ILV was in use that shows the two marker files (where the file contents have not been changed so On Disk Size is just the 4k for the extended attributes), and a deletion marker file.



ViewReparse.exe

This is also a home grown tool that I modified from some sources I found online many years ago. I'm not sure about the original copyrights so I'm not publicly posting my version. Maybe you can find something similar online.

The purpose of the tool is to open up and display the reparse tag and block information from a directory or file. This information is used by certain drivers, and the tag value maps to the name of a specific driver so this can be helpful.

For example, when looking at a folder in our redirection area, we might see this:

```
Administrator: Command Prompt
C:\windows\system32>\\nuc2\installers\Tim\ViewReparse\ViewReparse.exe
Syntax: ViewReparse filename


C:\windows\system32>\\nuc2\installers\Tim\ViewReparse\ViewReparse.exe C:\Users\Admin\AppData\Local\Package
s\BadAssTest_xwfzvwzp69w2e\LocalCache\Local\Microsoft\WritablePackageRoot\VFS\ProgramFilesX64\BadAssTest
TAG: [Microsoft] WCI
   Tag=80000018 DataLen=56 Reserved=0
   20 00 00 80 56 00 00 00 01 00 00 00 02 00 00 00    . . . V . . . . . . . . . . . .
   88 8d 71 82 d4 74 e6 4f 86 b4 a5 a1 3d bf 34 52  . . q . ╘ t µ O . . . . . . 4 R
   3c 00 56 00 46 00 53 00 5c 00 50 00 72 00 6f 00  . . V . F . S . \ . P . r . o .
   67 00 72 00 61 00 6d 00 46 00 69 00 6c 00 65 00  g . r . a . m . F . i . l . e .
   73 00 58 00 36 00 34 00 5c 00 42 00 61 00 64 00  s . X . 6 . 4 . \ . B . a . d .
   41 00 73 00 73 00                                A . s . s .


C:\windows\system32>
```

The tab indicates that it is the Wcifs mini-filter driver that handles the directory. Further data in the reparse block, which unfortunately is formatted on a per driver basis and generally not documented, seems to show the package path to the equivalent folder (in a form that is relative to the package root folder which would need to be known through other means).

This is why I feel that, with ILV, bindflt might be responsible for redirecting from the package to the redirection while wcifs is responsible for the reverse direction.

This odd combination I find problematic, and I am sure over the years we'll find an edge case where it becomes important!

BadAssTest.exe

This is an application written specifically for my needs in testing the ILV, but is flexible enough to be of help in any file based testing of this sort.

> *It should be obvious from the app name that either TMurgent doesn't have an HR department or that Tim doesn't listen to them.*

Source for this app has been added as a project under my GitHub Repository at https://GitHub.com/TimMangan/BadAssTest.

It is a Win32 Console application that will make certain filesystem calls to certain paths. It is controlled by an external text file that defines file paths and test scenarios to run. If you run the exe without a configuration file present it will dump out syntax help information like this:

C:\Windows\System32\cmd.exe

```
C:\Users\admin\Desktop>badasstest.exe
BadAssTest!
===========
BadAssTest is configured using a file named 'BadAssTest.txt' located in the same folder as the exe.
The file should consist of one or more lines of text.
Each line identifies a path to a file, followed by a comma and then a list of comma separated TestNumbers that
should be performed.
No quotation marks, escapes, or unnecessary spaces should be used.
The supported TestNumbers are:
        1 - std::filesystem::exists(...)
        2 - GetFileAttributes(...)
        3 - CreateFile for Read Only of only an Existing File; without reading contents.
        4 - CreateFile for Read/Write of only an Existing File; without writing contents.
        5 - DeleteFile(...)

        6 - FindFirstFile(path to a directory)

        7 - CreateFile for Read/Write of only an New File; with writing small content.
An example file:
        C:\Program Files\BadAssTest\Subfolder\TestFileA.ini,1,2,3
        C:\Users\Tim\AppData\Local\BadAssTest\Subfolder\TestFileB.cfg,1,3
        <Root>\VFS\ProgramFilesX64\BadAssTest\Subfolder\TestFileA.cfg,1,2,3,4,5
        <WritablePackageRoot>\VFS\Local AppData\BadAssTest\Subfolder\TestFileB.cfg,1,3,4
        <Root>\VFS\ProgramFilesX64\BadAssTest\Subfolder\TestFileY.cfg,7
        <Root>\VFS\ProgramFilesX64\BadAssTest\Subfolder,6
        <WritablePackageRoot>\VFS\Local AppData\BadAssTest\Subfolder,6
================= End of all tests =================


C:\Users\admin\Desktop>
```

I put this exe file, along with a configuration file and other additional package files, and a shortcut to run the exe into an MSIX package.

With this I can test the package without help, or by modifying the package to include the ILV, the PSF MFR, or both as needed. (I used my TMEditX tool for that package editing work).

When the app runs a specific test, in addition to outputting the test scenario and results to the screen, the app uses the registry to place markers at the start and end of the test (including test results). This makes it very easy to understand what test produces what events in a trace from a tool like Process Monitor.

An example of console output is shown here:

C:\Program Files\WindowsApps\BadAssTest_3.0.0.1_x64__xwfzvwzp69w2e\VFS\ProgramFilesX64\BadAssTest\BadAssTest.exe

```
BadAssTest!
===========
Starting Test C:\Program Files\BadAssTest\TestFileA.txt,1,2,3,4,5
Parsed Path C:\Program Files\BadAssTest\TestFileA.txt
Starting Test 1
        exists: SUCCESS

Starting Test 2
        GetFileAttributes: SUCCESS=0x20

Starting Test 3
        CreateFile(Read): SUCCESS

Starting Test 4
        CreateFile(Write): FAIL=0x3

Starting Test 5
        DeleteFile: FAIL=0x3

-------------------------Test is done.--------------------------

Starting Test C:\Users\Admin\AppData\Local\BadAssTest\TestFileB.txt,1,2,3,4,5
Parsed Path C:\Users\Admin\AppData\Local\BadAssTest\TestFileB.txt
Starting Test 1
        exists: FAIL=0x3

Starting Test 2
```

Combining this with Process Monitor can be very powerful for investigation, as shown in the following section.

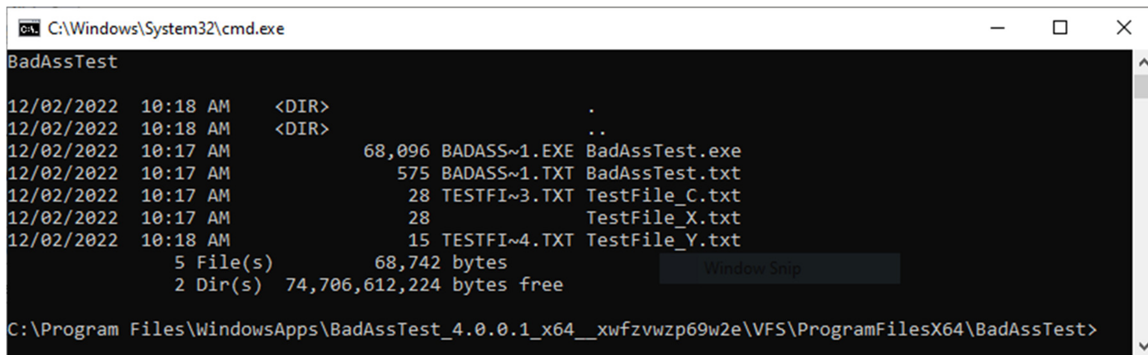
Invoke-CommandInDesktopPackage

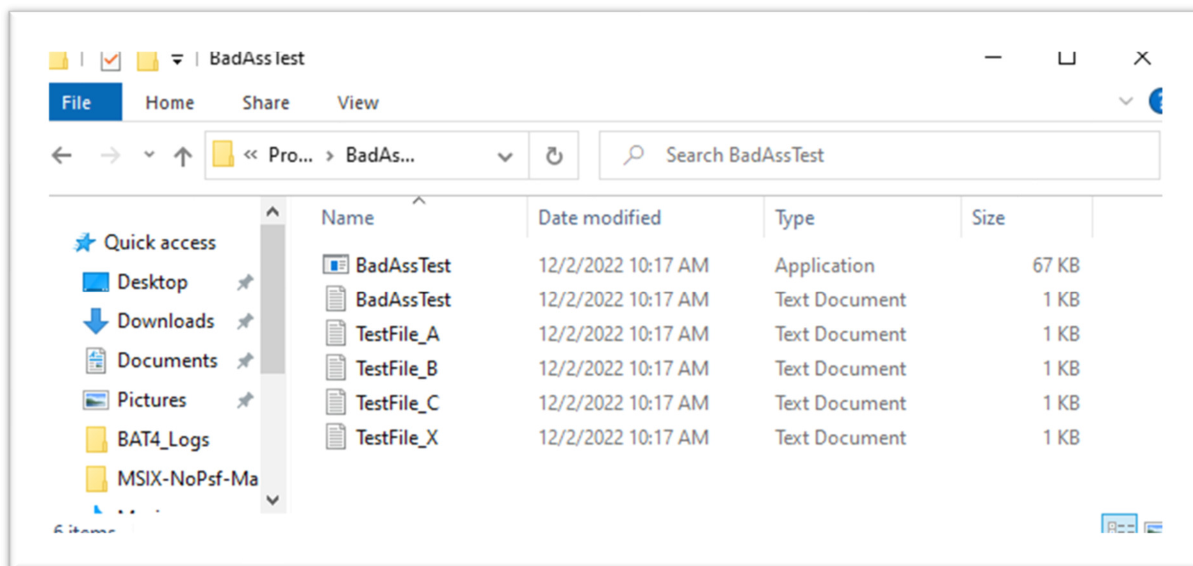
This PowerShell cmdlet can be used to start a process, like cmd or explorer, inside the container. This is useful to get a cmd prompt or explorer process running inside the container. These processes will automatically get the benefit of the ILV, and depending on the settings of the config.json file in the package, possibly that of the FileRedirectionFixup or MfrFixup.

This is important for debugging, because explorer or cmd running outside the container does not receive the benefit of ILV or the PSF fixups.

The cmdlet requires the PackageFamilyName, plut the AppId of one of the Applications from the package. It actually doesn't matter which AppId you pick. There are several tools around to help you with this so you don't have to find and type all that in.

For example, here are two screen captures taken after several file operations were made against the package (The "A" and "B" files were part of the original package and deleted, while the "X" file was not part of the package but added by the app). The cmd was run inside of the package, showing exactly what the app would "see", while the explorer is external to the package.

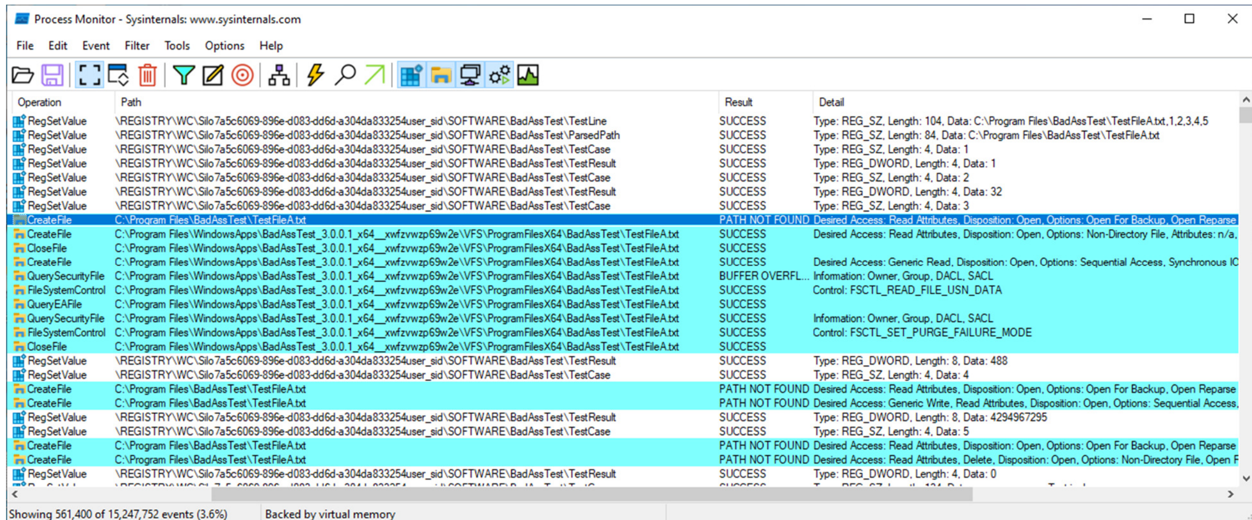




Hint: Use Process Explorer to show dlls loaded in a process to know if the PSF fixup is loaded in your test process!

BadAssTest and Advanced use of Process Monitor

For example, in the screen shot below we can see the start of processing a line from the file (the setting of the TestLine value on the top line. The ParsePath shows the filename to be used in this test after parsing for <Root> or <WrtiablePackageRoot> as part of the file path. TestCase shows the start of the test case (and type number which maps to a type of action). The TestResult is written after the test number is completed.

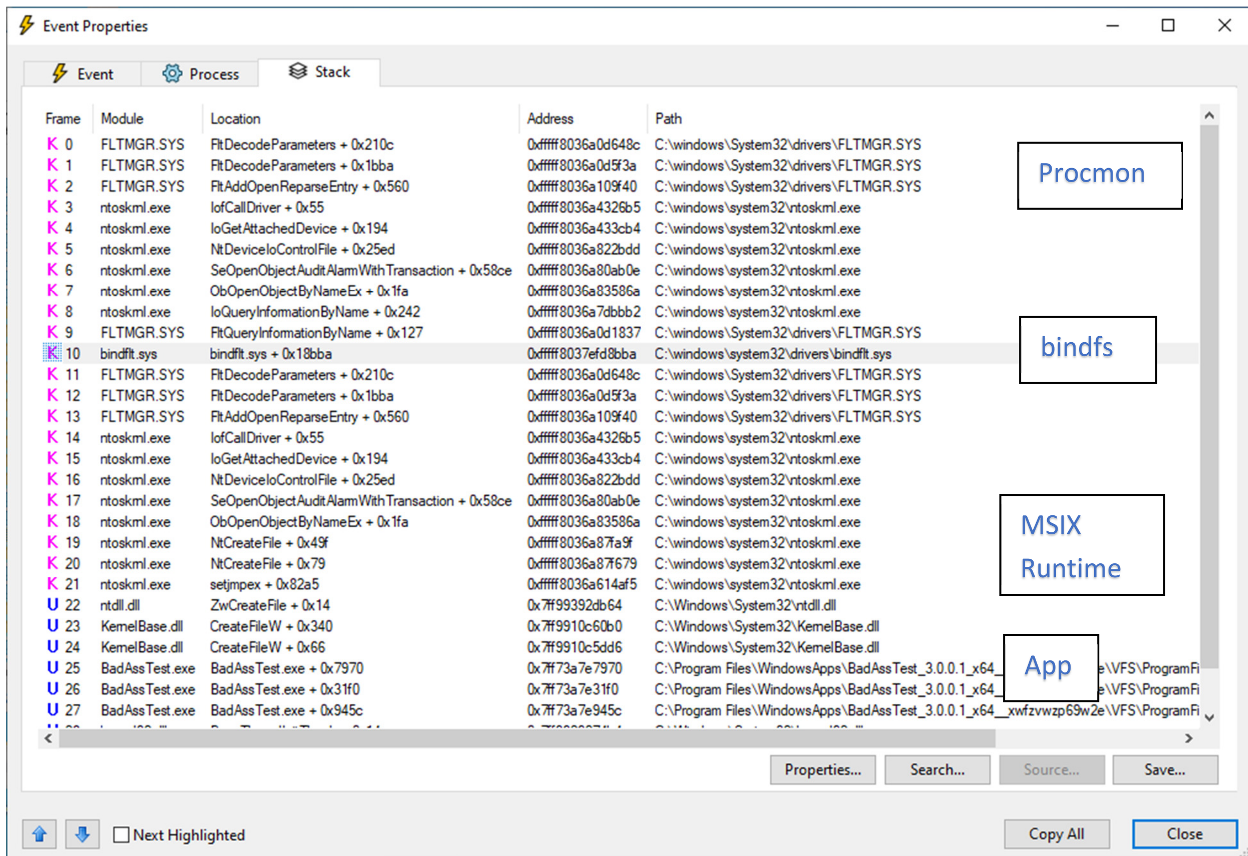


In the example you can see that:

- Test type 1 and type 2 produced no file events. In this case, Procmon was loaded below wcifs and the calls never got down to the ProcMon mini-filter driver. (Other tests implicate the binflt driver for stopping the calls and returning success).
- Test 3, 4, and 5, show events as highlighted.

Comparing procmon traces captured at different altitudes of the exact same test allows for comparison to determine the impact that a specific filter driver has on the operation.

Furthermore, by analyzing the stack shown on an event, we can learn even more. Double-clicking on a Process Monitor event line brings up a popup dialog with more detail. Part of that detail is the Stack tab. If you have read this far, you probably have had enough schooling to know what a program stack is about, so this can be helpful to us.



You read this with the application at the bottom, with each entry above a call made by the function below, so the visual is upside down from our filter altitude diagram previously. The Windows cache manager would be "above" this drawing.

We can see that the app (line 25) in user space called a flavor of CreateFileW, which in turn called a different flavor of CreateFileW (with different arguments), then ZwCreateFile which transitioned into kernel mode (setjmpex is an interrupt handler used for the transition and sets up the IRP used in the file i/o call. The IRP is what is passed through the mini-filter drivers up to the file system). Lines 15-11 are the entry to a mini-filter driver that wants to affect this call and we see on line 10 that it is in the binfs driver. Lines 4-0 are the calls leading into the Procmon Driver which performed the capture.

Furthermore, we can compare this stack to that of the next event for more information. Typically you might see that all of the addresses starting from the bottom of this list are identical up to a certain point. This implies (not proves) that the first event made its way to the filesystem and back to the level of the last stack line that did not change. For example, if in the next event the calls were identical from the bottom to and including bottom most NTCreateFileEx, we can assume that the app made a single call and NTCreateFileEx made two calls into the filesystem from there.

This can be useful in not only understanding where functionality is implemented, I have found this useful in reporting bugs to the correct vendor!

TMEditX

Version 3.0 of TMEditX (soon to be released) has support for ILV. It will default to recommending ILV as a fix for all packages. This version will also have support for MFR (without the ILVAware features). A pre-release of this software was invaluable for testing.

LiveKD/WinDebug

Such tools can also prove valuable, especially with checked builds of the OS. If you aren't actually debugging your own driver, however, it is a lot of work for little value.

__end__