Windows Component-Based Servicing (CBS): An In-Depth Overview

Understanding the Intricacies of the Windows Image Building Process

pivotman319

6th June 2024

Table of Contents

Introduction

Preface

This document aims to thoroughly explain how *Microsoft Windows* operating system images are assembled through the component-based servicing stack and also provides preliminary context on how the assembly of the respective components are prepared beforehand.

Acknowledgements

I would like to thank *Scamdisk* (nee *Mintel*), Anri Guramovich (aka *Applegame12345*), Gustave Monce (aka *gus33000*), Arminder Singh (aka *amarioguy*) and *WitherOrNot* for their help in guiding me around CBS internals over the course of two years.

I would also like to especially thank the fine individuals over at the *Anomalous Software Deterioration Corporation* (*ASDCorp*; formerly *Gamers Against Weed*) for providing the additional resources and tools I need to help thoroughly cover this subject.

Disclaimer

For this write-up, ***only* desktop-based Windows versions will be covered**. I will not be covering other Microsoft operating systems for the following reasons:

Admittedly, I do not have enough knowledge on how *Windows Phone* images are built. Documentation for these operating systems are **extremely limited** as Microsoft has kept quite a lot of things under wraps (including subjects which are *supposed* to be covered by the *Windows Phone Adaptation Kit*, but for some reason aren't), which makes it hard for myself and others to easily understand how to decipher the way other Windows releases are built. I do know for a matter of fact that such releases use a completely different staging process (which use cabinet files containing individual operating system components) and that such images are instead built with their own custom tooling in place.

These releases happen to be built by using the `ImgGen` tool, which was first put into play midway through *Windows Phone 8* development; sometime around Windows NT builds 80xx-814x (from the `FBL_CORE1_MOBILE_DEV` development branches) is where I'm hazarding that they eventually ultimately phased out `BootableSku`/WinSxS-based WP images. As far as I am aware, this tool is often still used by Microsoft to assemble WP-like images, such as *ModernPC* (`WCOSCDG`; aka *Windows 10X*), *`MobileCore`*/*`OneCoreUpdateOS`*, *Windows Holographic* (`AnalogOneCore`), et cetera.

Image Building Process

Post-Build Binary Spew, Windows Foundation and Preparation

Daily Build Cycle

Every day, Microsoft compiles Windows builds in a variety of CPU architectures and build flavors, which include Free (retail/consumer), Checked (debug) and previously Code Coverage, which included symbols as part of binaries (if I recall correctly – coverage builds are no longer being compiled today). These builds additionally target every SKU within the source tree.

For instance, one could have an `arm32chk ProfessionalWMC` build that targets the French (France) (`fr-FR`) localization, or an `x86fre ServerDatacenter` build that targets the English (United Kingdom) (`en-GB`) localization. The way these components are first bundled together happens during the post-build process, where binaries containing the raw component structure of what would eventually become a Windows image are placed in an output folder - more specifically, the Windows component store (`WinSxS`), which contains the raw, unorganized set of components that will be used to form a base Windows image. In unstaged Windows images, this store is simply referred to as just the `packages` directory.

This process is not exclusive to just *one* build. It happens across practically every different development branch that exists within the `OS` Git source tree. Component manifests (which may contain vital information such as required registry values, *NTFS* file permissions, binary file locations, *Base64*-encoded *SHA-1*/*SHA-256* file hashes) are first generated during the post-build process, followed by the update package manifests that contain the required metadata to install such components and their associated digitally-signed security catalogs, which contain signatures for binaries defined by said update manifest.

Certificate Chains

Depending on the conditions of the overall build's compile (such as the residing development branch and the way it is configured), security catalogs may be signed by the following certificate authority chains:

- *Microsoft Windows Production PCA 2011*: production code signing. Since Win7 SP1/mid-Win8 dev;
- *Microsoft Windows Production PCA 2023*: Ditto. Introduced during early *Gallium* (v24H1) development, adopted in Windows 11 2024 Update (v24H2) for EFI boot managers only – first existed alongside shift to proper certs for Secure Boot code signing (*Microsoft UEFI CA 2023*) in response to continued occurrences of original equipment manufacturers losing private keys used to sign existing EFI firmware;
- *Microsoft Windows Verification PCA*: production code signing; intended for WHQL driver verification, but was used in builds from `winmain_win8m1` and `winmain_win8m2` development branches;
- *Microsoft Windows PCA 2010*: pre-release code signing – used in `winmain_win8m3` and `winmain_win8beta`;
  - Pre-release code signing was also adopted in debug Windows builds (from *Windows 8 Release Preview* onwards) as well as within partner-only *Windows*

8.1 development branches (including, but not limited to `FBL_PARTNER_OUTXX`, `FBL_PARTNER(_QC/NV/INTELSOC)`)

- *MSIT Test CodeSign CA [#]*: test code signing, or;
- *Microsoft Development PCA 2014* (flight code signing; since Win10 *TH1*).
    - Note: certain branches also used a separate Development PCA certificate that utilized pre-release code signing in place of flight code signing, e.g.: **`rs_prerelease_prs`**, **`rs1_release_prs`**, **`winmain_prs`**, et cetera.
    - The Windows cryptography library (**`mincrypt`**, statically linked to in **`CI.dll`** and **`bootmgr/bootmgfw.efi`**) was altered during *Windows 10 Creators Update* development (in `RS_PRERELEASE` build 14965) to allow expired flight-signing certificates in an effort to prevent legitimate customers from unknowingly installing Insider builds that would have eventually stopped working.
- *Microsoft Windows OEM Root 2017*: OEM/partner-specific code signing certificate allowing hardware manufacturers to sign their own custom-built components for testing purposes only – actively used by Qualcomm. Was refreshed once to push chain expiration date further into the future.

Foundation Image Creation

The main gist of the whole assembly process boils down to three important factors:

- the *Windows Componentization Platform* (and the main CBS servicing stack)
- the Windows foundation image, containing the absolute dependencies needed to build the image, and;
- the aforementioned component and update package manifests (including the raw component binaries).

Windows foundation images are created by respectively calling C++ functions `CreateNewWindows()` and `CreateNewOfflineStore()` in the servicing stack library `WCP.dll`, short for the *Windows Componentization Platform*:

```
HRESULT CreateNewWindows(
    DWORD dwFlags,
    LPCWSTR szSystemDrive,
    POFFLINE_STORE_CREATION_PARAMETERS pParameters,
    PVOID *ppvKeys,
    DWORD *pdwDisposition
);

HRESULT CreateNewOfflineStore(
    DWORD dwFlags,
    POFFLINE_STORE_CREATION_PARAMETERS pParameters,
    REFIID riid,
    IUnknown *ppStore,
    DWORD *pdwDisposition
);
```

Disclaimer: As of Windows 10 Anniversary Update (RS1/v1607). These functions' variables may have been changed in later Windows NT versions.

Microsoft uses a tool officially dubbed the *Trusted Installer Client Console* (`CBSS`) to build the Windows image, which would call these functions in a separate tool and then perform the staging process. The foundation image may be placed inside of a virtual hard disk image that will contain the actual OS image. Further servicing operations made after initial neutral image staging process (such as language pack and feature installation) will eventually resort to image servicing operations made through automation by passing commands over to the *Deployment Image Servicing and Management Tool* (*`DISM`*).

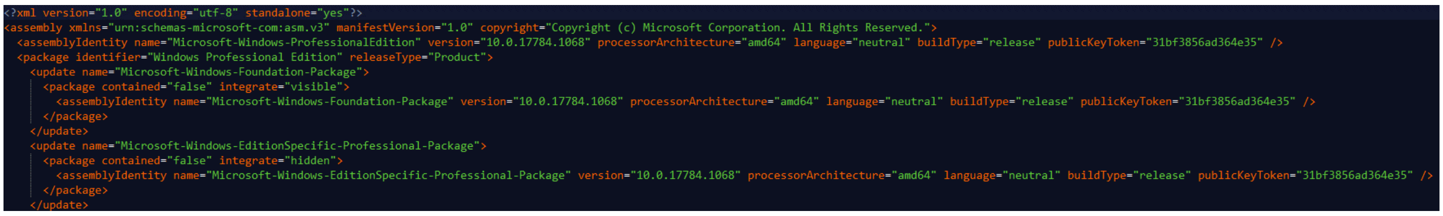
Resolving Packages

Package names in the Windows component store are structured under the following format:

`PackageName~PublicKeyToken~cpuarch~lang-REGION~1.2.3.4`

- | = Package name (e.g.: `Microsoft-Windows-StepsRecorder-Package`)
- | = Public key token (e.g.: `31bf3856ad364e35`) – bound to package/component signer
- | = CPU architecture (e.g.: `x86`, `arm64`, `mips`, `ia64`)
- | = Language and region type (e.g.: `en-US`; omitted in neutral packages)
- | = Package version (e.g.: `10.0.10586.1000`)

Update package manifests are divided into a subset of packages and deployment manifests to install into the operating system image:



```xml
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v3" manifestVersion="1.0" copyright="Copyright (c) Microsoft Corporation. All Rights Reserved.">
  <assemblyIdentity name="Microsoft-Windows-ProfessionalEdition" version="10.0.17784.1068" processorArchitecture="amd64" language="neutral" buildType="release" publicKeyToken="31bf3856ad364e35" />
  <package identifier="Windows Professional Edition" releaseType="Product">
    <update name="Microsoft-Windows-Foundation-Package">
      <package contained="false" integrate="visible">
        <assemblyIdentity name="Microsoft-Windows-Foundation-Package" version="10.0.17784.1068" processorArchitecture="amd64" language="neutral" buildType="release" publicKeyToken="31bf3856ad364e35" />
      </package>
    </update>
    <update name="Microsoft-Windows-EditionSpecific-Professional-Package">
      <package contained="false" integrate="hidden">
        <assemblyIdentity name="Microsoft-Windows-EditionSpecific-Professional-Package" version="10.0.17784.1068" processorArchitecture="amd64" language="neutral" buildType="release" publicKeyToken="31bf3856ad364e35" />
      </package>
    </update>
```

The operating system image is built by parsing a specific XML file (bearing the **`.mum`** file extension), an example being the "edition"/"product" manifest pictured above. The XML file is first checked for whether any specific conditions are met, such as the presence of a specific language pack, a core Windows feature or a Windows update; if there are no conditions present in the XML file, this step is skipped entirely. The servicing stack then parses the XML file, loading in any defined update package manifests and component "deployment" manifests which act as redirectors for the actual system component manifests.

Specific criteria for how feature installation is handled exists within the componentization platform - the servicing stack checks the value of the **`releaseType`** variable is equivalent to **`Product`** (a Windows edition) or **`Feature Pack`**; if the value matches the former, then the servicing stack performs a check to see if more than one Windows edition is being installed at once, and raises error code `CBS_E_MORE_THAN_ONE_ACTIVE_EDITION` if these conditions end up being true. This does not get thrown in cases where one or more editions are staged at once, but only one edition is actually installed on the operating system image.

The servicing stack attempts to locate such manifests by using a specific naming system that distinguishes components from another through the derivation of the architecture, component name, public key token, build number and localization values. This, in turn, generates a sort of "unique key" that is used by CBS to automatically locate component manifests, which is appended to the end of the actual component name:

```
amd64_Microsoft-Windows-OS-Kernel-MinWin_31bf3856ad364e35_10.0.17784.1068_none_e10b1ea85233711
```

- 🟫 = CPU architecture
- 🟦 = Component name
- 🟩 = Public key token
- 🟧 = Component version
- 🟪 = Language and region
- 🟥 = Derived "unique key"; an amalgamation of the five values

A shorter version of the above format, named "Winners", is also used for quickly associating and pairing components together by omitting the version field:

```
amd64_Microsoft-Windows-OS-Kernel-MinWin_31bf3856ad364e35_none_d0b24db91fb68e77
```

- 🟫 = CPU architecture
- 🟦 = Component name
- 🟩 = Public key token
- 🟪 = Language and region
- 🟥 = Derived "unique key"; an amalgamation of the four values

Feature Packaging

Microsoft may decide to package certain operating system features (such as the *Remote Server Administration Tool* (*RSAT*) or the Windows N edition-specific *Media Feature Pack*) into *LZMS*-compressed cabinet archives. These are often used for Windows updates, separate feature installation (such as the .NET Framework, the on-demand package of which is also made available online or built into the Windows installation media in the **`sources\sxs`** directory) or through online feature-on-demand installs initiated by the *Optional features* Settings subpage.

Typically, these cabinet files contain the needed subset of components + binaries to install the package. The main component update manifest is promptly renamed to **`update.mum`** to allow the CBS servicing stack to properly locate and install the respective component, although since Windows 10 Fall Creators Update, a copy of the update manifest with the original component name is also included as part of the package itself. Package installation is handled by decompressing the cabinet file into a separate temporary scratch directory located within the component store (or manually defined by the user by passing the **`/ScratchDir`** parameter via DISM), and then processing the update manifests using the staging and feature installation phases described later in the document.

Pre-Staging

Pre-staging begins immediately after the Resolve phase, which is the discovery and population of components that have yet to be installed. Component manifests (including deployment redirectors) are placed into the `Windows\WinSxS\Manifests` directory and are additionally compressed using Windows Update `DCM` compression (a variation of the *LZMS* compression algorithm), packed in the form of a `PA30` null-delta update binary (wherever applicable; the servicing stack **specifically excludes certain manifests from being compressed** such as those pertaining to Windows Common Controls to prevent a `CRITICAL_PROCESS_DIED` bugcheck from occurring).

The manifests are also placed inside of a central "manifest cache" found in the `Windows\WinSxS\ManifestCache` directory to allow the servicing stack to quickly load in manifests from a single binary rather than needing to perform disk read operations across multiple files in random order in an effort to optimize the overall image servicing process.

Components are then populated into the Windows image's registry in both the `COMPONENTS` and `SOFTWARE` registry hives, which were initially automatically generated in the form of empty registry hives by WCP function `CreateNewWindows()` and later populated by the `OfflineStore` mechanism. Update manifests and security catalogs are first copied over to the `Windows\servicing\Packages` directory; the aforementioned manifests' security catalogs (which, as mentioned earlier, are signed with either a SHA-1/SHA-256 certificate chain depending on the Windows version being dealt) are also placed within the following directories:

- `Windows\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}`
- `Windows\WinSxS\Catalogs*`

*Note: security catalogs in this directory are automatically renamed into their own SHA-256 hashes.

Staging and Component Installation

After the pre-staging process is completed, the servicing stack then begins to stage the actual update packages by parsing the components' manifest files and copying their associated binaries over to the root of the Windows component store, located at **`Windows\WinSxS`** – these components usually have an associated directory containing the required binaries which will later be placed into the final resulting image. The actual components' binaries (as well the respective component manifests) are first checked against matching *SHA-1*/*SHA-256* hashes located within both the manifest itself and within the provided update package manifest's security catalog and then copied into their respective locations. If no applicable hash is found for the respective binaries or component manifest, the servicing stack throws a critical error and immediately attempts to abort the operation.

After all components have been staged, the servicing stack runs three different types of installers in order:

- Primitive Installers (initializes basic registry and file permissions, allowing files to be hard-linked in the soon-to-be resulting Windows image)
- Midground Installers (Provides Advanced Installer-level capabilities for existing components – used for basic setup of functionality like core Windows system services)
- Advanced Installers, which pre-configures the resulting neutral image (as defined in the **`type`** directive for each component manifest). Also responsible for staging and installing drivers in the CBS Driver Store (located within the **`Windows\System32\DriverStore`** directory) using the same mechanisms used to install drivers via the built-in Windows Device Manager, as invoked by the **`drvStore`** advanced installer.

After the prerequisites have been fulfilled, the servicing stack then begins to hard-link every binary defined within each component manifest; for instance, the operating system kernel may not actually exist as **`Windows\System32\ntoskrnl.exe`**, but is instead a two-way shortcut for a singular operating system component which also happens to originate from the SxS store. Registry values and access control lists (within the operating system's NTFS partition and inside the Windows registry) are also applied against the respective resources where applicable.

Individual directories' file maps are then automatically generated for each hard-linked binary in the form of a CDF data file and placed inside of the **`Windows\WinSxS\FileMaps`** directory. The servicing stack may attempt to copy a specific component's binaries *in full* (and *not* hard-linked) over to the **`Windows\WinSxS\Backup`** directory, acting as a last-resort fallback for offline repair operations if it so happens that there exists corruption within the Windows component store.

At this point, the image can now be declared fully staged; however, there are some implications which need to be fulfilled before the operating system image can be properly used, which is covered in the next section.

Final Steps

Language Pack Installation

Neutral Windows images lack a language pack by default. In this state, the operating system cannot boot at all due to missing multilingual user interface binaries (MUIs) and will instead bugcheck with code `MUI_NO_VALID_SYSTEM_LANGUAGE`. This bugcheck can also occur due to product policy restrictions that are ultimately checked against by the Windows kernel. Some Windows editions or product keys only allow a specific (subset of) language(s) to be installed into the image; [1] one specific edge case would be the China government-mandated Windows editions, which primarily include:

- `CoreCountrySpecific` (Windows Home China)
- `ProfessionalCountrySpecific` (Windows 10/11 Pro China Only)
- `EnterpriseG` (Windows 10 Enterprise China Government Edition (CMGE))
- `EnterpriseGN` (Ditto; N variant)

The user must request the servicing stack to install their desired language pack (which comes packaged in the form of a cabinet file containing the respective components; these are essentially installed in the same way as the staging process described above) and configure it as necessary to ensure that certain aspects of the operating system (such as UI elements and branding resources) are displayed properly in front of the user. Certain language packs contain accommodations tailored towards specific regions or countries, such as those originating from Asia or the Middle East, and may differ visually (such as the font choice or element sizes; the Dutch language pack, for instance, has its own icon sizing variables, whereas the Korean (South Korea/Republic of Korea) language pack utilizes its own font for the *Windows Aero* visual style named *Microsoft YaHei*).

Windows Setup Unattended Datafiles

Each Windows edition may come with its own unattended data file, which contain specific customizations such as crash dump settings, Windows Explorer search configuration, licensing data or power settings. This data file is copied over to the `Windows` root operating system directory and then subsequently applied by DISM and/or Windows Setup through the `OEMDefaultAssociations` servicing stack plugin, located in the `Windows\System32` directory.

Language and Region, Image Serviceability State and SetupCl

Finally, the user then requests the servicing stack to set the default UI and system language that will be used across the entire image. The servicing stack attempts to mark the image as fully serviceable by adjusting the `SOFTWARE` and `SYSTEM` hives, flagging the system image with `IMAGE_STATE_GENERALIZE_RESEAL_TO_OOBE` in the registry and in `Windows\Setup\State\State.ini` (the latter being required in modern-day Windows releases) and preventing the operating system image from being modified by DISM whenever the user attempts to invoke `DISM /Cleanup-Image /StartComponentCleanup /ResetBase`. The operating system image is also modified to require launching the `setupcl` utility with specific servicing operation flags (as defined in the `SYSTEM` hive) to allow the image to be properly serviced by the Windows servicing stack during the second phase of the

Windows installation process (usually referred to as the hardware detection phase), which generally involves detecting hardware present on the user's device and installing applicable system drivers. Since Windows 8, provisioned AppX applications are also configured by the servicing stack on both a system- and user-facing level.

The final resulting operating system image is produced, and may then (optionally) be compressed into a Windows Imaging Format file (`.wim`) or distributed in the form of a pre-configured virtual hard disk (`.vhd`) image.

Amendments Made Since Windows 7

Although the overall servicing process has remained largely unchanged since Windows 7, Microsoft have made a number of underlying operating system changes that further warranted some minor alterations to the existing component-based servicing stack. These changes are described below:

Windows Runtime-specific servicing stack changes

Windows 8 introduced the Windows Runtime (WinRT) application model, an unmanaged application runtime that leverages Component Object Model (COM) API calls in place of older library imports/exports. Applications based on this model primarily run within an isolated "sandbox" or within a "silo" (as of Windows 10 RS2) to close out potential security holes – such applications therefore have their own subset of file, directory and process permissions that are completely separate from the usual permissions laid down within other areas of the Windows operating system.

Such an application is usually packaged inside of a Zip64 binary that contains the metadata and binaries required to install and use it, otherwise known as *AppX*. To account for these changes, new APIs for AppX package management (specifically `AppxDeploymentClient/Server`, `AppxPackaging`) were introduced, and the *Windows PowerShell* side of the Windows servicing stack was further updated to include a new servicing provider simply named `Appx`, which offers functionality for a then-new subset of commandlets that can install, provision, or obtain a list of AppX application packages – these notably include `Add-AppxPackage`, `Add-ProvisionedAppxPackage`, `Remove-AppxPackage`, and `Get-AppxPackage`.

Staged Windows images usually do not come with any AppX applications by default – the sole exceptions of course being those that are needed to allow the OS shell to work, which are marked as "system apps". These applications cannot be uninstalled at all, as removing them would ultimately break core operating system functionality such as the web-based out-of-box experience (`Microsoft.Windows.CloudExperienceHost`; introduced in Windows 10 *TH1*), the Start menu (`Microsoft.Windows.StartMenuExperienceHost`), the Immersive Control Panel (aka *PC settings* in Windows 8.x, or just *Settings* in Windows 10 onward), and/or the Action Center.

Operating system images can be configured to include or exclude the ordinary subset of inbox AppX applications; server operating system images are *always* excluded from these operations. Since Windows 8.1 client, images are pre-configured to automatically install these applications by default, although edition-specific exemptions apply.

Windows editions that are manually configured to not install inbox applications include:

- All applicable *Windows Server* editions, including their Server Core counterparts (`ServerWeb`, `ServerStandard`, `ServerDatacenter`, `ServerHyperCor`, `ServerAzureCor`, `ServerAzureStackHCICor`);
- `ProfessionalS(N)` – Scrapped Windows 10 Pro LTSB servicing release;
- `EnterpriseS(N)(Eval)` – Windows 10/11 Enterprise LTSC (N) (Evaluation);

- `EnterpriseG(N)` – Windows 10 Enterprise CMGE

Some localization and/or edition combinations also have a pre-defined list of applications that are excluded from installation. A Windows N edition coupled with an installed Korean language pack may exclude both media *and* messaging applications to comply with anti-trust/fair competition legislation that has since been signed into law.

Windows 10 Language Features on Demand (FoDs)

Since the initial Windows 10 release (*Threshold 1*/v1507), specific language pack features (which include text-to-speech, basic natural language selection (NLS) data, optical character recognition (OCR) and inking/handwriting) have been separated out into their own neutral on-demand feature packages, and must therefore be installed alongside the language pack to allow specific system features to work properly. These features include (but are not limited to) the *Windows Ink Workspace* (first introduced as part of the *Windows 10 Anniversary Update* (*Redstone 1*/v1607)), the *Narrator* accessibility tool and the now-defunct *Cortana* service.

Temporary Switch to DISM for Image Servicing

Microsoft made a temporary change that slightly impacted Windows 7, 8.x and early Windows 10 by using DISM to stage the operating system in place of the earlier Trusted Installer client console. Evidently, this was reverted for unknown reasons, although it is assumed that the switch back to **`CBSS.EXE`** was likely done due to architectural changes related to feature-on-demand packages introduced as part of Win10 v1507.

Conclusion

Microsoft's method of building Windows operating system images is rather unconventional, and can be often compared to how Linux distributions such as Fedora, Ubuntu or Debian are packaged – Fedora's example, being that the user is offered a choice to pick what components they can install via **`dnf`** (in the form of grouped packages that play a role in what is essentially forming a complete "environment"), bears many similarities to the way Windows itself handles "updates", as certain individual components (such as the Windows Server WoW64 compatibility layer) are also grouped together in a single handy package. Ubuntu, on the other hand, offers its own package management system over a command-line interface with options to directly install and uninstall features on demand via the **`apt`** utility.

I hope this document helps adequately explain the basic and advanced fundamental concepts of how the Windows side-by-side component store works internally. Again, I would like to reiterate and extend my thanks to the people largely involved in providing me with the resources needed to aid in the creation of this document as a whole (as described within the above acknowledgements included in the Introduction section).

Thank you for reading.

-pivotman319

Citations

1. *awuctl* – Licensing Stuff: Product Keys (published on 12th April 2023)