Post Build Signing

In publishing v3, signing will be moved out of the build step and will be completed post-build in a release pipeline. This reduces the time required for builds that do not require signing and allows repos to have more control over when the signing process occurs.

Any repositories that wish to move to post-build signing should ensure that they're using v3 publishing, including Publish.proj. Repositories that produce wix installers (msi's, wixlabs, etc) are additionally required to use Microsoft.DotNet.Build.Tasks.Installers targets and ensure that ItemsToSign in the AssetManifest includes the generated *wixpack.zip files generated by the SDK. If PostBuildSign==true then no other steps are needed, post-build signing is enabled and configured by default.

Steps to ensure that post-build signing is correctly set up for a repo: A manifest file is created as part of the build process, it can be found in AssetManifests/Manifest.xml. This file lists everything that will be signed once the build is complete. If this file has been created and the contents of the ItemsToSign section match the expected set of signed files for the build then no further action is needed.

Post Build Signing Technical Details

When performing normal in-build signing, the build process generates a list of files to sign (e.g. nupkg files, dlls, msi's, etc.) during the build process. These are combined with information about what certificates to use for each file type or specific file name (located in Sign.props) and fed to SignTool. SignTool processes the input files and performs signing via ESRP, replacing the input files in place with the signed versions. Depending on the build scenario, SignTool may be called multiple times within a build to iteratively build up some archives. For instance, it may be called on individual dlls which are then packaged into an msi, which is then passed to SignTool.

Performing post build signing is more involved. The drop that is to be signed is comprised of the outputs of many repos. Some of those outputs may already be signed, some of those outputs may have been repackaged within the outputs of other builds, etc. Thus, most of the effort in post build signing is related to two aspects of the process: - **Deconstructing (and reconstructing) archives** - With in-build signing, SignTool knew how to deconstruct zip and nupkg archives. These could simply be unzipped, their contents replaced in-line, then zipped back up. Post build signing requires dealing with msi installers too. These are not as amenable to changes to contents (e.g. the msi database encodes file lengths, which change with signing). - **Tracking the source of all the files** - Let's say dotnet/runtime notes that apphost.exe should not be signed, then the installer repackages that binary in a zip file. When post-build signing runs, it needs to be able to identify that apphost.exe, which appears to be located within an installer produced zip file, actually came from runtime and thus should not be signed.

Post build signing deals with this by recording sidecar information at build time, then gathering the closure of that info when post build signing runs. The additional information includes the following: - Archives that can be used to reconstruct installers produced by a build. - Manifest files which describe the signing data for a build (e.g. items to sign and file extension -> certificate mappings)

The process works as follows: 1. Gather the drop of a build that needs to be signed, including all un-released inputs builds. 2. Locate all build manifests (containing signing information for each input build) 3. Build a list of all files to sign (files to sign should be found in the drop), and the specific certificate info required for those files/extensions (and any files within containers). Each input item to sign is associated with the certificate info present in the same build. 4. Pass this data to SignTool 5. In SignTool: 1. Walk each item to sign. If the item is an archive, unpack it and begin to process the contained files. 2. After a file is unpacked (if necessary), determine what signing cert and/or strong name it should get (if any). Add info to a map of filename+hash of file -> info. 3. Continue until all files are unpacked. 4. Starting at the most nested set of files, create a Microbuild compatible project file and submit for signing. 5. Repack those files into their enclosing containers, if any. 6. Repeat, moving upwards in the tree of files to sign until all files have been repacked. 6. Upload files to storage (feeds and blob storage).

Determining certificates

The crux of the SignTool algorithm is determining the certificate for a specific file. *Side note: Certificates are associated with a specific file name or file extension, though they cannot be associated with a specific path. For instance, 'foo/bar/baz.zip' is only dealt with as 'baz.zip'.* When dealing with a single repo, SignTool just interprets the signing info as applying to all files passed to it. A baz.zip found anywhere in the input file set is treated identically. When dealing with post build signing, it's possible that a foo.dll found in an sdk asset may actually come from the runtime, which may have indicated that *.dll get a different certificate.* So SignTool needs to track where foo.dll originally came from*. Did it come from the build of dotnet/runtime, or from dotnet/sdk? To do this, SignTool associates each input top level item to sign and the signing info for a build with a 'collision priority ID' This collision priority ID serves as a way to disambiguate the source signing information for a nested artifact. As each top-level asset is unpacked, the hash of each nested file is determined and looked up in a map. If the map already contains a file with the same name and hash, the collision priority ID's are compared. If the current one in the map is lower, the signing info for the existing file is used. Otherwise, the certificate information is recomputed based on the lower collision priority ID.

Post Build Signing a single repo vs full product

Post build signing is intended to sign a full set of product inputs. Given that some repos produce artifacts that repackage artifacts from other repos, the full graph of builds and dependent builds are required to *fully* sign artifacts. If you only have the artifacts from an individual repo, then post build signing will sign what it can, given the information available. If it is repackaging an MSI from another repo, and that repo's manifest / metadata are not available, then post build signing will sign the packaged artifact, but not its contents because it is unable to repackage an MSI without the light command package data from that dependent repo. The result will be that signing validation would fail and properly report unsigned files. It is up to the repo owner to investigate whether those unsigned files are valid, invalid, or should be ignored because they are the result of post build signing with partial input (false positives).

Additional Information

More information about the file types that are signed and the certificates used to sign them can be found here: https://github.com/dotnet/arcade/blob/master/src/Microsoft.DotNet.Arcade.Sd

Additional information about the publishing process in general can be found here: https://github.com/dotnet/arcade/blob/master/Documentation/CorePackages/Publishing.md

Additional information about the in-build signing process can be found here: https://github.com/dotnet/arcade/blob/master/Documentation/CorePackages/Signing.md

Was this helpful?