# Instead of putting a hash in the Portable Executable timestamp field, why not create a separate field for the hash?

devblogs.microsoft.com/oldnewthing/20240815-00

August 15, 2024


Raymond Chen

Some time ago, we learned why the module timestamps in Windows 10 are so nonsensical: Because they aren't timestamps any more. They are a hash of the resulting binary.

But why not invent a new field called, say, `UniqueValue` for the hash, rather than putting it in the timestamp field?

> https://t.co/iPc0RdM9vc
> yes, stupid decision imho; could use a diff. field for that
>
> — Adam (@Hexacorn) February 15, 2024

Well, for one thing, that would be a breaking change. If you take a binary produced by a linker that puts the hash in a new field and run it on an older system, the older system will ignore the hash and use the timestamp, so the hash does nothing.

But wait, why are we gathered here in the first place? The reason for using a hash instead of a timestamp is to permit reproducible builds, and the Wikipedia page specifically calls out timestamps for scorn:

> According to the Reproducible Builds project, timestamps are "the biggest source of reproducibility issues."

If you put a timestamp in the binary, then it's no longer reproducible: Making no changes and rebuilding will produce a different binary because the timestamp will be different.

If we want a reproducible build, we simply have to get rid of the timestamp.

Remember what the timestamp is used for: It's used by the module loader to detect whether precalculated addresses of imported functions should be trusted: When the addresses are precalculated (by binding), the timestamp of the module that was used as the basis of the precalculation is recorded by the importing module. When the loader loads the importing module, it checks whether that timestamp matches the timestamp recorded in the module from which the functions are being imported. If it matches, then the precalculated values are used. If it doesn't match, then the precalculated values are ignored and new values are calculated from scratch

Okay, so maybe we can use some other source as the timestamp, rather than the timestamp of the build itself. How about the timestamp of the most recent commit?

That still doesn't work because you can build multiple binaries from the same source code. Any precalculated values from a debug build will not be correct for a release build, and vice versa. Any switches that affect code generation must change the timestamp because the resulting binary is different and in particular the addresses of exported functions may change.

Okay, but maybe we can start with the timestamp and, say, hash the compiler switches into a 16-bit value that gets added to the original timestamp. That way, you still get a pseudo-timestamp that is within a day of the actual timestamp.

But now you've swung the pendulum too far the other way. Previously, the problem was that the timestamp didn't change when it should have. Now the problem is that the timestamp changes when it didn't need to. Maybe you made a commit to a `README.md` file. This isn't even part of the source code, but it'll change the "most recent commit" timestamp. Okay, so maybe you look only at commits that modify source code. But now you add a new `enum` to a header file (say, `windows.h`) that is included by every component, but only one component actually takes advantage of it. The change to the header file will update the "most recent source code commit" timestamp of every component, even though only one of the components actually changed as a result of the new `enum`. The other components are binary identical, or would be if it weren't for the timestamp.

The way to get the timestamp to change when the binary changes, but only when the binary changes, is to make the timestamp depend only on the binary itself (minus the timestamp field).[1]

**Bonus chatter**: Making the timestamp a hash of the binary contents simplifies the process of determining which binaries were affected by a change: Look for binaries whose timestamp hashes changed. Not only does this make things easier for the servicing team (to identify which binaries need to be included in the next monthly update), it's also handy as part of your regular workflow: If you change a header file with the intention of fixing an issue in one component, and several dozen files changed timestamps, then that's a signal that what you thought was a change with very limited scope turned out to have a much larger scope than you thought, and maybe you should figure out what unintended consequences your change precipitated.[2]

[1] This is a tautology, but sometimes it helps to state the tautology explicitly.

[2] One common example of this is adding a new method to a COM interface. This causes the IID to change, which in turn causes every module that produces or consumes that interface to change. What you thought was a simple change to one binary ended up pulling a dozen binaries into the next monthly patch. Instead, you should create a new interface for your new method and leave the original interface alone.