Why are the module timestamps in Windows 10 so nonsensical?

devblogs.microsoft.com/oldnewthing/20180103-00

January 3, 2018

Raymond Chen

One of the fields in the Portable Executable (PE) header is called `TimeDateStamp`. It's a 32-bit value representing the time the file was created, in the form of seconds since January 1, 1970 UTC. But starting in Windows 10, those timestamps are all nonsense. If you look at the timestamps of various files, you'll see that they appear to be random numbers, completely unrelated to any timestamp. What's going on?

One of the changes to the Windows engineering system begun in Windows 10 is the move toward *reproducible builds*. This means that if you start with the exact same source code, then you should finish with the exact same binary code.

There are lots of things that hamper reproducibility. One source is the language itself. For example, anonymous namespaces may not have a programmatically-accessible name, but since the objects within it have external linkage, they need to have a name nonetheless, and the name must be different for different source files. How does it ensure the names are unique? Does the compiler use a random number generator to generate these names? Is it a hash of the file name?

Another source is the compiler's internal code generation algorithms. For example, if a compiler chooses between two optimizations depending how much RAM is available, or how powerful the processor is, then that prevents the result from being reproducible because two systems with different hardware configurations may end up producing different outputs. Or if the optimizer has a failsafe switch that abandons an operation if the algorithm is still running after 500ms. Or if the optimizer uses a non-deterministic register allocation strategy. Or if the compiler uses a deterministic algorithm ("sort all local variables") but uses a non-determinstic criterion ("… by the heap address of the data structure we use to keep track of each variable.").

There are also inputs to the system outside the compiler that hamper reproducibility. For example, the full path to the file being compiled will show up in `__FILE__` preprocessor directives, which will cause problems when built from different machines with different names for the root directory that holds the source code. (Or even from the same machine with two copies of the source code.) There may be files auto-generated by the build process that go into the compiler (for example, the output of compiler-compilers); those need to be deterministic too.

Timestamps are another source of non-determinism. Even if all the inputs are identical, the outputs will still be different because of the timestamps.

Okay, at least we can fix the issue with the file format. Setting the timestamp to be a hash of the resulting binary preserves reproducibility.

“Okay, but why not set the file timestamp to the the timestamp of the source code the binary was created from? That way, it’s still a timestamp at least.” That still breaks reproducibility, because that means that touching a file without making any changes will result in a change in binary output.

Remember what the timestamp is used for: It’s used by the module loader to determine whether bound imports should be trusted. We’ve already seen cases where the timestamp is inaccurate. For example, if you rebind a DLL, then the rebound DLL has the same timestamp as the original, rather than the timestamp of the rebind, because you don’t want to break the bindings of other DLLs that bound to your DLL.

So the timestamp is already unreliable.

The timestamp is really a unique ID that tells the loader, “The exports of this DLL have not changed since the last time anybody bound to it.” And a hash is a reproducible unique ID.

Raymond Chen

Follow