Manipulating LastWriteTime without leaving traces in the NTFS USN Journal

gtworek.github.io/PSBits/lastwritetime.html

As we all know, each file and folder on the NTFS volume has its own metadata/attributes, including the `LastWriteTime` - set automatically when file write occurs. The `LastWriteTime` is relatively easy to read both by Windows Explorer GUI, commandline (in the `dir` output) or within PowerShell by issuing `(dir X:\temp\test.txt).LastWriteTime` command. Regardless management needs, such attribute may be priceless when it comes to the Forensics or Incident Response processes. Finding all files written within some date/time boundaries will significantly narrow the area for an investigation. Of course, bad actors know about it, and they set the `LastWriteTime` to some value from the past, or just the original one, when it comes to files present on the volume during the attack. Everyone can set this attribute, and the technique is well known: it relies on `SetFileInformationByHandle()` API call and may be achieved for example through PowerShell with simple `(dir X:\temp\test.txt).LastWriteTime = "1999-12-12"` etc. Detection of such manipulation is possible both when it happens (scenario I am not covering here) and later - thanks to the data collected in the NTFS USN Journal. The journal contains a reliable log describing events related to the volume content change, and (obviously) it includes `LastWriteTime` change as well. It may be displayed with `fsutil.exe usn ReadJournal` command and clearly identified by the reason presented as "*Basic info change*".

```
Usn              : 127563952
File name        : test.txt
File name length : 16
Reason           : 0x00008000: Basic info change
Time stamp       : 2022-03-24 16:50:38
File attributes  : 0x00000080: Normal
File ID          : 0000000000000000002000000027be8
Parent file ID   : 000000000000000000870000000b55a4
Source info      : 0x00000000: *NONE*
Security ID      : 0
Major version    : 3
Minor version    : 0
Record length    : 96
```

A query against a NTFS USN Journal can help identify manipulated files with same efficiency and accuracy as checking `LastWriteTime` does. And it is a common practice during properly performed investigations. Now it should be obvious, why the technique allowing to change file without updating its `LastWriteTime` may be very useful for bad actors. A quick digression about API and low-level programming of file writes: to write a file, three steps are usually performed:

1. `CreateFile()` - API call taking the file name and returning a *handle* to it. Despite its name it not only creates a file. Opening the existing one (does not matter if the purpose is to write or just read) will require `CreateFile()` anyway.
2. `WriteFile()` - API call, taking the *handle* obtained in the previous step plus some data, writing the data to the file.
3. `CloseHandle()` - API call to let the OS know that the work is finished.

If there is a need to change `LastWriteTime`, yet another API call is used: `SetFileInformationByHandle()` consuming a handle obtained from `CreateFile()`, the purpose (`FileBasicInfo` in this particular case), and dates/times to be set. As the call requires a valid handle, it should happen somewhere in between `CreateFile()` and `CloseHandle()`. The date/time to be passed is *64-bit value representing the number of 100-nanosecond intervals since January 1, 1601 (UTC)* and may be calculated using API functions (such as `SystemTimeToFileTime()`) or even simpler: read and stored before file write operation, then applied after it, leading to a file with unchanged `LastWriteTime`. The very simple code could look like this:

```
handle = CreateFile(filename, ...);
GetFileInformationByHandleEx(handle, FileBasicInfo, oldDateTime, ...);
WriteFile(handle, dataBuffer, ...);
SetFileInformationByHandle(handle, FileBasicInfo, oldDateTime, ...);
CloseHandle(handle);
```

Even as the file is modified, the `oldDateTime` stores the original (before write happened) set of dates and times, and the same set is re-applied just after the write operation, leaving the file with dates/times looking just like before the operation. When we look at the journal, we can observe the following set of entries:

1. `0x00000002: Data extend`
2. `0x00008002: Data extend | Basic info change`
3. `0x80008002: Data extend | Basic info change | Close`

It means the file metadata was modified, which is relatively easy to spot. And what if we do something looking stupid at the first sight: read the original datetime, write it (unchanged!) and then modify the file content? Like in the code below:

```
handle = CreateFile(filename, ...);
GetFileInformationByHandleEx(handle, oldDateTime, ...);
SetFileInformationByHandle(handle, FileBasicInfo, oldDateTime, ...); //Set datetime
FIRST
WriteFile(handle, FileBasicInfo, dataBuffer, ...); //Modify file NEXT
CloseHandle(handle);
```

It looks pointless, as the write operation should change the `LastWriteTime` and no one fixes it later, but it works! If you set the date/time BEFORE the content change, specifying the original one, (or -1), the subsequent file content changes using the same handle will NOT

update the datetime and of course will not leave any trace of updating it in the journal. Journal will contain only:

1. `0x00000002: Data extend`
2. `0x80000002: Data extend | Close`

I have never seen it documented, but it works. I can also observe some places where it is used by the Windows itself, which may mean the described behavior is 100% intentional.

Some final thoughts:

1. Be careful when relying on undocumented features, as YMMV.
2. If you want to play on your own, I would suggest to use separate volume (as the C drive is quite busy and the journal may report hundreds of entries per second) and create a journal on it with `fsutil.exe usn createJournal X:`
3. You can observe journal entries 'live' with `fsutil.exe usn readJournal X: wait tail`

Of course, any file modification will leave own journal entry, even if the date/time change event is not stored. In practice, DFIR investigators very rarely care about it, because it is too common. The typical approach relies on hunting for date/time change in the journal and then checking files, assuming metadata was not manipulated if the journal contains no evidence of such operation. It leads to a conclusion that your procedure for finding modified files should be revised and potentially updated.