The malware analyst’s guide to PE timestamps

0xc0decafe.com/malware-analyst-guide-to-pe-timestamps/

January 22, 2021

This blog post is all about time. More exactly, timestamps found in Portable Executable (PE) files that describe a (possible) compilation date. These PE timestamps may even reveal details about a threat actor. For instance, it is possible to deduce a threat actor’s working hours and use this information – hopefully together with other artifacts – for attribution purposes. But be aware: even though PE timestamps can be a valuable forensic artifact, they can be forged with ease. While the COFF header field `TimeDateStamp` is the most-known place to look for a compilation timestamp, there are more places where we can find timestamps in a PE file.

But there are also special cases where PE timestamps are not correct due to new build features, linker bugs, or simply forging. We’ll look into these cases and see how we can still get some information about the compilation date. Afterward, we’ll learn how to read PE timestamps with a wide range of tools and how to automate this work in order to deal with larger datasets. Of course, there are several real-world applications for malware / cyber threat intelligence analysts of what we’ll learn in this blog post. Finally, I’ll give you some ideas how you can utilize PE timestamps to reveal certain behaviors or characteristics of threat actors.

This article assumes a basic understanding of the PE format. Good beginner write-ups are "An In-Depth Look into the Win32 Portable Executable File Format" and "Portable Executable File Format". As a follow-up, Ange Albertini wrote a much more detailed technical article on the PE format.

This work wouldn't have been possible without many great blog posts and papers published by the infosec community. I am standing on the shoulders of giants! I refer to these posts and papers in the following whenever possible.

The COFF File Header field TimeDateStamp (_IMAGE_FILE_HEADER)

The probably most cited field of the PE format that holds a compilation timestamp is the field `TimeDateStamp` of the COFF File Header. It is a `DWORD` (32 bit / 4 bytes) member of the struct `_IMAGE_FILE_HEADER` / COFF File Header. This struct is defined as follows:

```
typedef struct _IMAGE_FILE_HEADER {
 WORD  Machine;
 WORD  NumberOfSections;
 DWORD TimeDateStamp;
 DWORD PointerToSymbolTable;
 DWORD NumberOfSymbols;
 WORD  SizeOfOptionalHeader;
 WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

`TimeDateStamp` is in epoch, which measures the seconds that have passed since January 1, 1970 UTC. Since it is a DWORD value, its range is quite limited. This will lead – if not fixed before – to an integer overflow in the year 2038. Furthermore, epoch **should not be** confused with *Win32 epoch*, which starts in the year 1601.

As part of the linking process of the final PE binary, the linker sets the field `TimeDateStamp` in the COFF File Header. However, its purpose today is mainly supporting the module loader as Raymond Chen lays out in "Why are the module timestamps in Windows 10 so nonsensical?":

> 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." […]
>
> *Raymond Chen – Why are the module timestamps in Windows 10 so nonsensical?*

Bottom line: even though nobody had the intention to tamper with a timestamp, it *may already be unreliable* in certain cases.

Additional timestamps found in the PE file format

In addition to the field `TimeDateStamp` of the COFF File Header, there are further places where we can find possible compilation timestamps in PE files.

I think the most complete listing was compiled by Walied Assar. He lists additional places in the PE format where we can find possible compilation timestamps. I'll refer only to the ones that can be used in order to determine the compilation time in the following sections.

Nevertheless, you should have a look at the other possible fields that Walied Assar's aforementioned article mentions. Just for completeness, I'll list the directories that I **do not include** in the following:

- `TimeDateStamp` in `_IMAGE_IMPORT_DESCRIPTOR` (see `_IMAGE_EXPORT_DIRECTORY` ; used in bound imports)
- `TimeDateStamp` in `_IMAGE_BOUND_IMPORT_DESCRIPTOR` (see `_IMAGE_EXPORT_DIRECTORY` ; used in bound imports)
- `TimeDateStamp` in `_IMAGE_LOAD_CONFIG_DIRECTORY` (obsolete since Windows XP)

_IMAGE_EXPORT_DIRECTORY

The export directory (`_IMAGE_EXPORT_DIRECTORY`) of the PE format contains also a field called `TimeDateStamp` . Not all linkers seem to fill this field but Microsoft Visual Studio linkers fill it for both DLL and EXE PE files. For reference, the export directory structure `_IMAGE_EXPORT_DIRECTORY` is defined as follows:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions;
    DWORD AddressOfNames;
    DWORD AddressOfNameOrdinals;
  } IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

The field `TimeDateStamp` is relevant for bound imports as other PE files that use bound imports (see Import Binding) refer to the value of this field in their own field `TimeDateStamp` of `IMAGE_IMPORT_DESCRIPTOR` . Or in other words: the linker fills the field `TimeDateStamp` of `IMAGE_IMPORT_DESCRIPTOR` with the value of `TimeDateStamp` of

`_IMAGE_EXPORT_DIRECTORY` of the PE file that they bind their import to. This is the reason why the field `TimeDateStamp` of `IMAGE_IMPORT_DESCRIPTOR` are unusable to determine a possible compilation date.

The value of this field describes the time when the export directory was created. Therefore, it doesn't have to be equal to the `TimeDateStamp` from the COFF File Header. But it is often equal or very close to it.

_IMAGE_RESOURCE_DIRECTORY

Another `TimeDateStamp` field resides in the resource directory (`_IMAGE_RESOURCE_DIRECTORY`). The corresponding structure is defined as follows:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
     DWORD   Characteristics;
     DWORD   TimeDateStamp;
     WORD    MajorVersion;
     WORD    MinorVersion;
     WORD    NumberOfNamedEntries;
     WORD    NumberOfIdEntries;
 } IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

While Microsoft Visual Studio linkers seem not to set this field, (old) Borland linkers set it. This is especially relevant for malware that is compiled with a Delphi version between Delphi 4 and Delphi 2006. As we later see, there is a known bug so that these versions do not correctly set the field `TimeDateStamp` of the COFF File Header. However, they set it – if the PE file comprises resources – in the `IMAGE_RESOURCE_DIRECTORY` and its subdirectories.

_IMAGE_DEBUG_DIRECTORY

If the linker emits debug information (/DEBUG), then the PE file contains an array of debug directories (`_IMAGE_DEBUG_DIRECTORY`). The corresponding structure contains a field called `TimeDateStamp` and is defined as follows:

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
     DWORD   Characteristics;
     DWORD   TimeDateStamp;
     WORD    MajorVersion;
     WORD    MinorVersion;
     DWORD   Type;
     DWORD   SizeOfData;
     DWORD   AddressOfRawData;
     DWORD   PointerToRawData;
 } IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;
```

As a PE file can contain several `_IMAGE_DEBUG_DIRECTORY` s (stored in an array), it is worth to check all of them, in case the timestamp of the first seems to be forged.

Furthermore, one type of debugging information contains a timestamp. The field `Type` holds several constants (range `0x0` – `0x9`) that indicate the format of the debugging information. If the field is set to ***`IMAGE_DEBUG_TYPE_CODEVIEW`*** (`0x2`), then we can follow `PointerToRawData` of `_IMAGE_DEBUG_DIRECTORY` and we'll find the CodeView information.

The relevant structures `_CV_HEADER` and `_CV_INFO_PDB20` are defined as follows:

```
#define CV_SIGNATURE_NB10 '01BN'
#define CV_SIGNATURE_NB09 '90BN'

typedef struct _CV_HEADER
{
 DWORD dwSignature;
 DWORD dwOffset;
} CV_HEADER, *PCV_HEADER;

typedef struct _CV_INFO_PDB20
{
 CV_HEADER CvHeader;
 DWORD dwSignature;
 DWORD dwAge;
 BYTE PdbFileName[];
} CV_INFO_PDB20, *PCV_INFO_PDB20;
```

If the `_CV_HEADER`'s signature in `dwSignature` equals `CV_SIGNATURE_NB10`, then the PDB format is 2.0. In this case, the `dwSignature` contains an epoch value describing when the debug information was created. If it is not in PDB format, then `dwSignature` may comprises an unique identifier for each build. Note that the field `dwAge` is not a timestamp in epoch but rather an ever-increasing value that indicates an update to the PDB information.

Special cases

There are several noteworthy special cases where the timestamps are either not reliable or other timestamps should be preferred. The following sections will look into three of these cases and show possible ways to circumvent this limitation.

Delphi timestamps

Do you know what happened on 1992-06-19? I googled it and found out that this was the day Batman Returns was released in the USA. But that's only relevant for Batman fans and not really related to timestamps. Something different happened, at least, if you believe the PE timestamps of likely thousands or even millions of Delphi binaries. They all were built on `1992-06-19 22:22:17`. The timestamp value is `0x2A425E19` (`708992537`).

What sounds like a very productive programmer evening is actually a bug in many Delphi compiler versions. Delphi versions before Delphi 2007 (likely Delphi 4 – Delphi 2006) did not set the field `TimeDateStamp` at all. It seems that they just used a preconfigured value for

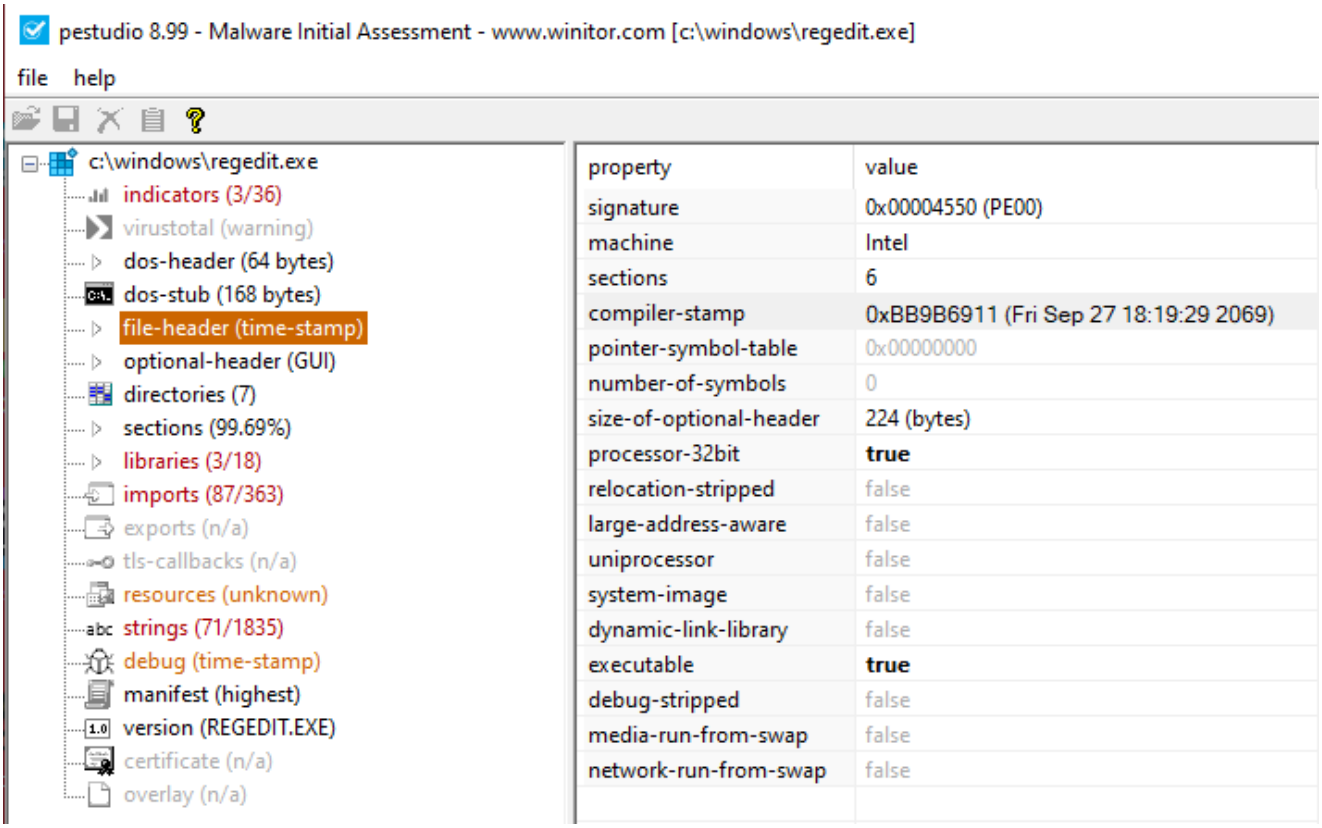
this field. Maybe they used a PE stub that they filled out accordingly. However, there is a way to determine the real compilation timestamp of these binaries. If the sample comprises a resources directory (`_IMAGE_RESOURCE_DIRECTORY`), we can read its `TimeDateStamp` field instead. But be aware that the value is not an epoch timestamp but a MS DOS timestamp.

Bottom line: if you see a binary with a timestamp value of `1992-06-19 22:22:17`, then you are likely dealing with a Delphi binary that was compiled with a Delphi version before Delphi 2007. If there are resources, then you may get at least a compilation estimate from the timestamp found in `_IMAGE_RESOURCE_DIRECTORY`.

Windows 10 reproducible builds timestamps

Windows 10 timestamps are nonsensical since Microsoft moved towards reproducible builds. In a nutshell, a reproducible build means starting a build from the exact same source code yields the exact same binary code. Obviously, timestamps are one obstacle in the way to reproducible builds. Therefore, timestamps are set to the hash of the resulting binary, which preserves reproducibility.

Let's have a look at a Windows 10 binary to verify this behavior. For instance, `regedit.exe` comprises a `TimeDateStamp` value `0xBB9B6911` as can be seen in the following screenshot (using the tool pestudio). If we translate this value to UTC time, then this date would be way in the future. As expected, the `TimeDateStamp` value of Windows 10 binaries is unreliable.



Inspecting the PE compilation timestamp of regedit.exe (Windows 10) with pestudio.

Even though I am not aware of a way to get a compilation timestamp directly from these binaries, there are two possible ways to get at least an approximation. First, if the file was scanned at VirusTotal, then it must be older than the first submission date. This may be a very inaccurate approximation but a good starting point.

Second, a better way to approximate the age of the binary would be to use a binary index like Winbindex. For instance, here we can check if it indexes the `regedit.exe` file from the previous example. And indeed: Winbindex lists this `regedit.exe` file with sha256 hash `d5de45decfd0fa08ac12f5725dfc7e5af1e3ed0325559101990fdcf02c439441` and states that it is part of Windows 10 1903 and 1909. So the earliest we can get is `2019-05-21`, which is the release date of Windows 10 1903.

Gozi2 / ISFB timestamps

Gozi2 / ISFB and its forks are wide-spread malware strains. Malware analysts are likely to stumble upon one or more of them during their career. Even though the timestamps of a Gozi / ISFB sample may be forged, there is a “feature” of this malware family and many of its forks that is likely more reliable. As stated by Maciej Kotowicz in his report on ISFB:

> One of first operations of all ISFB' variants is to decode strings that are stored in .bss section, section that is normally used to keep non-initialized global variables. The algorithm used for string encoding is quite simple, it is a rolling xor with a compilation date as a key.
>
> *ISFB: Still Live and Kicking by Maciej Kotowicz*

Again, the old rule of thumb holds here: threat actors are often as lazy as we are. Therefore, they will not touch this "feature". We can see this in an ISFB fork called LOLSnif. The sample that was observed in campaigns in April 2020 loaded the plaintext key "Apr 1 2020" to decrypt its strings.

```
; Attributes: bp-based frame

get_compilation_date_isfb proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8

push    ebp
mov     ebp, esp
sub     esp, 0Ch
push    esi
push    edi
mov     esi, offset aApr12020 ; "Apr  1 2020"
lea     edi, [ebp+var_C]
movsd
movsd
movsd
mov     eax, [ebp+var_C]
xor     eax, [ebp+var_8]
pop     edi
pop     esi
leave
retn
get_compilation_date_isfb endp
```

ISFB fork LOLSnif loads compilation date as plaintext string for string decryption

On the correctness of timestamps

Threat actors who compile their malicious binaries can easily tamper with any of the aforementioned timestamp values. I would say that forging timestamps is a low-hanging fruit and not doing so shows a lack of or disinterest in Operational Security (OPSEC), respectively.

As a rule of thumb: you should never blindly trust the timestamp found in any of the `TimeDateStamp` fields as they can easily be forged! However, you can increase the confidence that you have in the correctness of this value by, for instance, correlating this with intrusion dates obtained from incident response engagements or metadata obtained from sources like VirusTotal (e.g. `First Submission`). Furthermore, you should never trust the `TimeDateStamp` value of packed binaries as this is one of the PE header fields that packers tamper with in the first place. However, the `TimeDateStamp` values of unpacked binaries are in many cases (but certainly not all!) correct as threat actors often do not care about tampering with it.

Case study: custom loader DLLs used in SUNBURST attacks

For instance, the threat actor who is responsible for the sophisticated supply chain attack known as SUNBURST forged timestamps as stated in Microsoft's report "Deep dive into the Solorigate second-stage activation: From SUNBURST to TEARDROP and Raindrop":

> The custom loader DLLs dropped on disk carried compile timestamps ranging from July 2020 to October 2020, while the embedded Reflective DLLs carried compile timestamps ranging from March 2016 to November 2017. The presence of 2016-2017 compile timestamps is likely due to attackers' usage of custom Malleable C2 profiles with synthetic compile timestamp (*compile_time*) values. At first glance it would appear as if the actor did not timestamp the compile time of the custom loader DLLs (2020 compile timestamps). However, forensic analysis of compromised systems revealed that in a few cases, the timestamp of the custom loader DLLs' introduction to systems predated the compile timestamps of the custom loader DLLs (i.e., the DLLs appear to have been compiled at a future date).
>
> *Microsoft Security – Deep dive into the Solorigate second-stage activation: From SUNBURST to TEARDROP and Raindrop*

The interesting point here is that both the loader DLLs as well as the CobaltStrike Beacons have forged timestamps. The timestamps of the CobaltStrike Beacons were likely forged as part of the CobaltStrike framework and carried timestamps way in the past. However, the interesting thing here is that the outer layer – the custom loader DLLs – carried timestamps that laid in the future.

Even though it might be wild speculation why these timestamps predated the actual intrusion dates, one possible explanation is that the threat actor wanted the intrusion to appear to last less time than it actually lasted, in case the loader DLLs would have been found by the local network defenders.

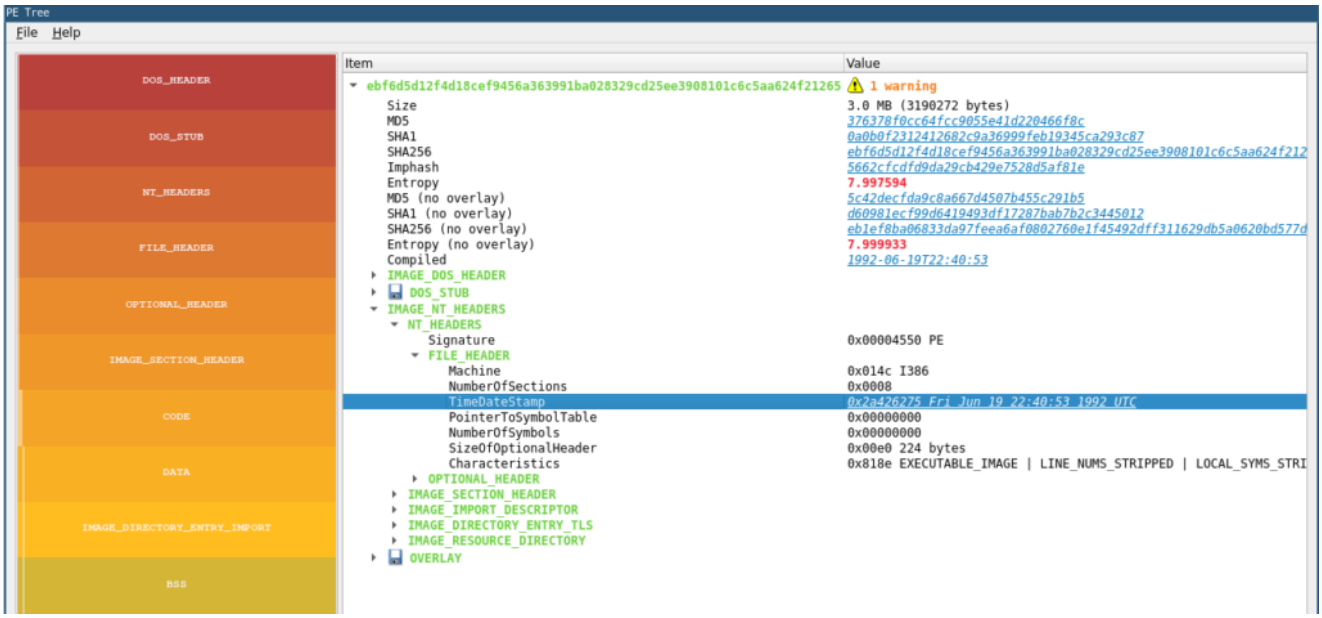
Read PE timestamps

Numerous tools and (Python) modules allow us to read (and modify) several fields of the PE header that can hold (compilation) timestamps. The following sections will show how these timestamps can be read with tools that I deem important. The first tools serve for manual inspection of individual PE files. But the later sections will show how to use two well-known Python modules in order to automate the PE timestamp extraction as a means of analyzing several PE files of a campaign / a threat actor.

Note that there is also an issue with several tools. While some display the TimeDateStamp as UTC time, some localize the TimeDateStamp and display incorrect information. IMHO the correct way is to display timestamps as UTC time and additionally indicate the shift that is required so that it's in the supposedly correct time zone.

Read PE timestamps with a PE file viewer

This one is a no-brainer. Every PE file viewer like, for instance, PE Tree, pestudio, or PE-bear displays the aforementioned `TimeDateStamp` fields as seen in the following screenshot using PE Tree. Note how PE Tree correctly displays the `TimeDateStamp` field as UTC time.



PE Tree displaying the field TimeDateStamp of the PE format's FILE_HEADER structure.

Read PE timestamps with TimeDateStamp

Walied Assar wrote a tiny command line program called TimeDateStamp. This command line program inspects half of a dozen timestamps that are defined in the PE format. Use the `-f` flag with a path to a PE file as follows:

```
C:\Users\USER\Desktop>TimeDateStamp.exe -f yara64.exe
     TimeDateStamp from _IMAGE_FILE_HEADER        ---> Fri Jun 26 08:37:17 2020
     TimeDateStamp from _IMAGE_EXPORT_DIRECTORY   ---> Empty
     TimeDateStamp from _IMAGE_DEBUG_DIRECTORY    --->
     TimeDateStamp from _IMAGE_LOAD_CONFIG_DIRECTORY ---> Empty
```

Another neat feature is that we can input a hex value (with or without leading `0x`) and the tool spits out the corresponding date. For instance, if we input `0x2A425E19` (do you still remember what that was?), then it spits out `Fri Jun 19 22:22:17 1992` correctly:

```
C:\Users\USER\Desktop>TimeDateStamp.exe 2A425E19
 Fri Jun 19 22:22:17 1992
```

Read PE timestamps with an hex editor

Daniel Plohmann's tweet with a list of DWORDs with only the most significant byte set and their corresponding dates is particularly interesting. For those who work frequently with HexEditors inspecting binary formats (not necessarily the PE format), this table is very helpful. Based on the most significant byte of the timestamp DWORD, we can quickly estimate the date.

For instance, the Delphi timestamp `0x2A425E19` falls in the bin `0x2A000000` and `0x2B000000`, or for humans `1992-04-30 14:11:12` and `1992-11-10 17:31:28`. Note that each bin covers more or less six months.

I've generated the bins for 2015 – 2025, which should be the most relevant as of 2021:

```
U -> 0x55000000 - 2015-03-11T09:42:40
 V -> 0x56000000 - 2015-09-21T15:02:56
 W -> 0x57000000 - 2016-04-02T19:23:12
 X -> 0x58000000 - 2016-10-13T23:43:28
 Y -> 0x59000000 - 2017-04-26T04:03:44
 Z -> 0x5a000000 - 2017-11-06T07:24:00
 [ -> 0x5b000000 - 2018-05-19T12:44:16
 \ -> 0x5c000000 - 2018-11-29T16:04:32
 ] -> 0x5d000000 - 2019-06-11T21:24:48
 ^ -> 0x5e000000 - 2019-12-23T00:45:04
 _ -> 0x5f000000 - 2020-07-04T06:05:20
 ` -> 0x60000000 - 2021-01-14T09:25:36
 a -> 0x61000000 - 2021-07-27T14:45:52
 b -> 0x62000000 - 2022-02-06T18:06:08
 c -> 0x63000000 - 2022-08-19T23:26:24
 d -> 0x64000000 - 2023-03-02T02:46:40
 e -> 0x65000000 - 2023-09-12T08:06:56
 f -> 0x66000000 - 2024-03-24T11:27:12
 g -> 0x67000000 - 2024-10-04T16:47:28
 h -> 0x68000000 - 2025-04-16T21:07:44
 i -> 0x69000000 - 2025-10-28T00:28:00
```

Read PE timestamps from the command line

There are many command line tools that can read the PE header. One tool I like to use is readpe from the pev toolkit. pev is open source, works on several platforms, and comes with many handy tools when dealing with PE files from the command line. Let's just use `readpe` to read the field `TimeDateStamp` of the COFF File Header from a packed IcedId PhotoLoader sample (4e7161be03f206c1b086bb15b47470ec1c9381302eb34d0e76915496aec77193, first submission to VT: 2020-05-27 23:31:27).

We call `readpe` with the `-h` flag, which allows us to specify a header that we would like to inspect. For our purpose, this is the COFF header (`coff`). As a result, `readpe` dumps all fields from this header.

```
readpe -h coff 4e7161be03f206c1b086bb15b47470ec1c9381302eb34d0e76915496aec77193
COFF/File header
     Machine:                        0x14c IMAGE_FILE_MACHINE_I386
     Number of sections:             4
     Date/time stamp:                1432567928 (Mon, 25 May 2015 15:32:08 UTC)
     Symbol Table offset:            0
     Number of symbols:              0
     Size of optional header:        0xe0
     Characteristics:                0x103
     Characteristics names
                                         IMAGE_FILE_RELOCS_STRIPPED
                                         IMAGE_FILE_EXECUTABLE_IMAGE
                                         IMAGE_FILE_32BIT_MACHINE
```

However, the timestamp is forged since the first submission of this sample was in Spring 2020 but the timestamp says that the sample is from Spring 2015. Let's unpack the sample with unpac.me and we receive one unpacked child (70fe6ccca21ce51800c7b998e8fc06997eac07e086f1dcdd87765b2dcea72395):

```
readpe -h coff 70fe6ccca21ce51800c7b998e8fc06997eac07e086f1dcdd87765b2dcea72395
COFF/File header
     Machine:                        0x14c IMAGE_FILE_MACHINE_I386
     Number of sections:             5
     Date/time stamp:                1586956141 (Wed, 15 Apr 2020 13:09:01 UTC)
     Symbol Table offset:            0
     Number of symbols:              0
     Size of optional header:        0xe0
     Characteristics:                0x102
     Characteristics names
                                         IMAGE_FILE_EXECUTABLE_IMAGE
                                         IMAGE_FILE_32BIT_MACHINE
```

Now, this timestamp looks much better. It is likely to be the correct compilation timestamp of this sample due to the first submission to VT differs just one month. Furthermore, there is a blog post published at the end of April 2020 that described this new variant of IcedId's PhotoLoader.

Read PE timestamps with pefile

pefile is a Python module to parse and modify PE files. The following script (based on this blog post) shows how to read the field `TimeDateStamp` from the COFF File Header and convert it to UTC time. In addition, it prints out how old the PE file is. Note that accessing other `TimeDateStamp` fields as described in Section "Additional timestamps found in PE files" works similarly. Please refer to the usage examples of pefile.

```
import datetime
import pefile
import sys

pe = pefile.PE(sys.argv[1])
ts = int(pe.FILE_HEADER.dump_dict()['TimeDateStamp']['Value'].split()[0], 16)
utc_time = datetime.datetime.utcfromtimestamp(ts)
t_delta = (datetime.datetime.today() - utc_time).days
print(utc_time.strftime("%Y-%m-%d %H:%M:%S +00:00 (UTC)") + f" ({t_delta} days old)")
```

If we run this simple script on a Delphi sample (Wabot, ebf6d5d12f4d18cef9456a363991ba028329cd25ee3908101c6c5aa624f21265), then we get a somewhat surprising result:

```
python get_timestamp_pefile.py
ebf6d5d12f4d18cef9456a363991ba028329cd25ee3908101c6c5aa624f21265
1992-06-19 22:40:53 +00:00 (UTC) (10436 days old)
```

The value is `0x2A426275`, which translates to `1992-06-19 22:40:53`. This is not the expected Delphi timestamp `0x2A425E19`. The most significant two bytes are equal (`0x2A42`), but the least significant two bytes / WORD is different. A possible explanation is that the timestamp was modified.

Read PE timestamps with lief

lief is another project to parse and modify PE files. Its core is written in C++ but there is a Python module as well. In contrary to pefile, lief is can parse various executable formats such as ELF and MachO while offering the same interface. Therefore, lief works well in projects that do not only target one platform. Furthermore, lief added several advanced features not yet implemented in pefile like working with PE Authenticode.

Read COFF File Header timestamps

The following code snippet shows the script `get_timestamp_lief.py` that reads the PE Header field `TimeDateStamp` (called by lief `time_date_stamps`) and converts it to UTC time (analogous to the pefile script of the previous section). Just import the module `lief` and open the binary with the `parse` method. The COFF File Header is accessible via `header` and the field `TimeDateStamp` is called `time_date_stamps` in lief.

```
import datetime
 import lief
 import sys
 binary = lief.parse(sys.argv[1])
 ts = binary.header.time_date_stamps
 utc_time = datetime.datetime.utcfromtimestamp(ts)
 t_delta = (datetime.datetime.today() - utc_time).days
 print(utc_time.strftime("%Y-%m-%d %H:%M:%S +00:00 (UTC)") + f" ({t_delta} days
old)")
```

Read Delphi timestamps

Again, Delphi binaries that were compiled with Delphi 4 – Delphi 2006 had a bug: all of them had the same `TimeDateStamp` value in the COFF File Header. However, there is a way (as described by Hexacorn) to get the compilation timestamp via the field `TimeDateStamp` in the resource directory (`_IMAGE_RESOURCE_DIRECTORY`). Be aware that in Delphi binaries this field does not hold an epoch timestamp but rather a MS DOS timestamp.

The following script reads the a possible compilation timestamp for Delphi binaries. It boroughs the function `from_msdos` from the project Time Decode to convert the MS DOS timestamp to a readable string. First, it checks if the `TimeDateStamp` field of the `IMAGE_FILE_HEADER` equals the expected Delphi timestamp (seen in Delphi 4 – Delphi 2006 binaries) in lines 26 – 27. In this case, it checks if there are resources with `binary.has_resources` in line 29 and then gets resource directory with `binary.resources` in line 30. Finally, it gets the timestamp and converts it from MS DOS format to a readable string.

```
import datetime
 import lief
 import sys

 def from_msdos(msdos):
     """taken from https://github.com/digitalsleuth/time_decode"""
     msdos = hex(msdos)[2:]
     binary = '{0:032b}'.format(int(msdos, 16))
     stamp = [binary[:7], binary[7:11], binary[11:16], binary[16:21], binary[21:27],
binary[27:32]]
     for val in stamp[:]:
         dec = int(val, 2)
         stamp.remove(val)
         stamp.append(dec)
     dos_year = stamp[0] + 1980
     dos_month = stamp[1]
     dos_day = stamp[2]
     dos_hour = stamp[3]
     dos_min = stamp[4]
     dos_sec = stamp[5] * 2
     if (dos_year in range(1970,2100)) or not (dos_month in range(1,13)) or not
(dos_day in range(1,32)) or not (dos_hour in range(0,24)) or not (dos_min in
range(0,60)) or not (dos_sec in range(0,60)):
         dt_obj = datetime.datetime(dos_year, dos_month, dos_day, dos_hour, dos_min,
dos_sec)
         return dt_obj.strftime('%Y-%m-%d %H:%M:%S')
     return "Not a valid MS DOS timestamp"

 binary = lief.parse(sys.argv[1])
 ts = binary.header.time_date_stamps
 if ts == 0x2a425e19:
     print('Found binary compiled with Delphi 4 - Delphi 2006')
     if binary.has_resources:
         root = binary.resources
         if root.time_date_stamp != 0:
             ts = root.time_date_stamp
             dos_time = from_msdos(ts)
             print(f'_IMAGE_RESOURCE_DIRECTORY: {hex(ts)} -> {dos_time}')
         else:
             print('Timestamp of _IMAGE_RESOURCE_DIRECTORY is set to zero')
     else:
         print('Binary has no resources.')
 else:
     utc_time = datetime.datetime.utcfromtimestamp(ts)
     print('Likely not a binary compiled with Delphi 4 - Delphi 2006')
     print(hex(ts) + ' -> ' + utc_time.strftime("%Y-%m-%d %H:%M:%S +00:00 (UTC)"))
```

I tested it with an older Azorult (version 3.1) sample:

```
> python lief_delphi_ts.py azor3.1_patched_fixed.pe32
Found binary compiled with Delphi 4 - Delphi 2006
 _IMAGE_RESOURCE_DIRECTORY: 0x4cbe1371 -> 2018-05-30 02:27:34
```

This looks a lot better. First, it matches the Delphi timestamp in the COFF File Header as expected. Luckily, there are resources and hence a `_IMAGE_RESOURCE_DIRECTORY` struct. The date `2018-05-30` seems to be a reasonable compilation timestamp for an Azorult version 3.1 sample.

What can we do with PE timestamps?

Now that we know where we can find PE timestamps and how we can read them using a wide range of tools, this section answers the final question: what can we do with PE timestamps? The following sections should inspire what we can do with them but also warn you once more that relying solely on this feature alone can certainly be very misleading.

Use case: Determine probable intrusion date

If timestamps found in PE files were always reliable, they could be used to determine probable intrusion dates, **especially if you are an observer on the outside**. For instance, it is common to hunt for ransomware samples on online cloud services like VirusTotal as these samples contain information that can identify the possible victim.

For instance, CL0P ransomware timestamps are reliable. Firstly, incident response engagements suggest this. Secondly, the temporal correlation between the sample timestamps, the first submission to VirusTotal, and the publication of the victim's data on their leak portal seem to be in line. If you combine these factors then you can likely estimate the deployment date of the ransomware from the outside.

Nevertheless, this is in theory not limited to ransomware deployments. PE timestamps of samples that were used in a certain intrusion (possible one in the interest of the public domain) can be utilized to estimate the intrusion dates of nation-state actors like Winnti. Samples of Winnti typically contain information such as a C&C domain and a campaign ID that can be used to guess possible victims. In addition, PE timestamps seem to be reliable to a certain degree to estimate at least the year of the intrusion.

Use case: Timeline of threat actor's campaigns and threat actor's working hours

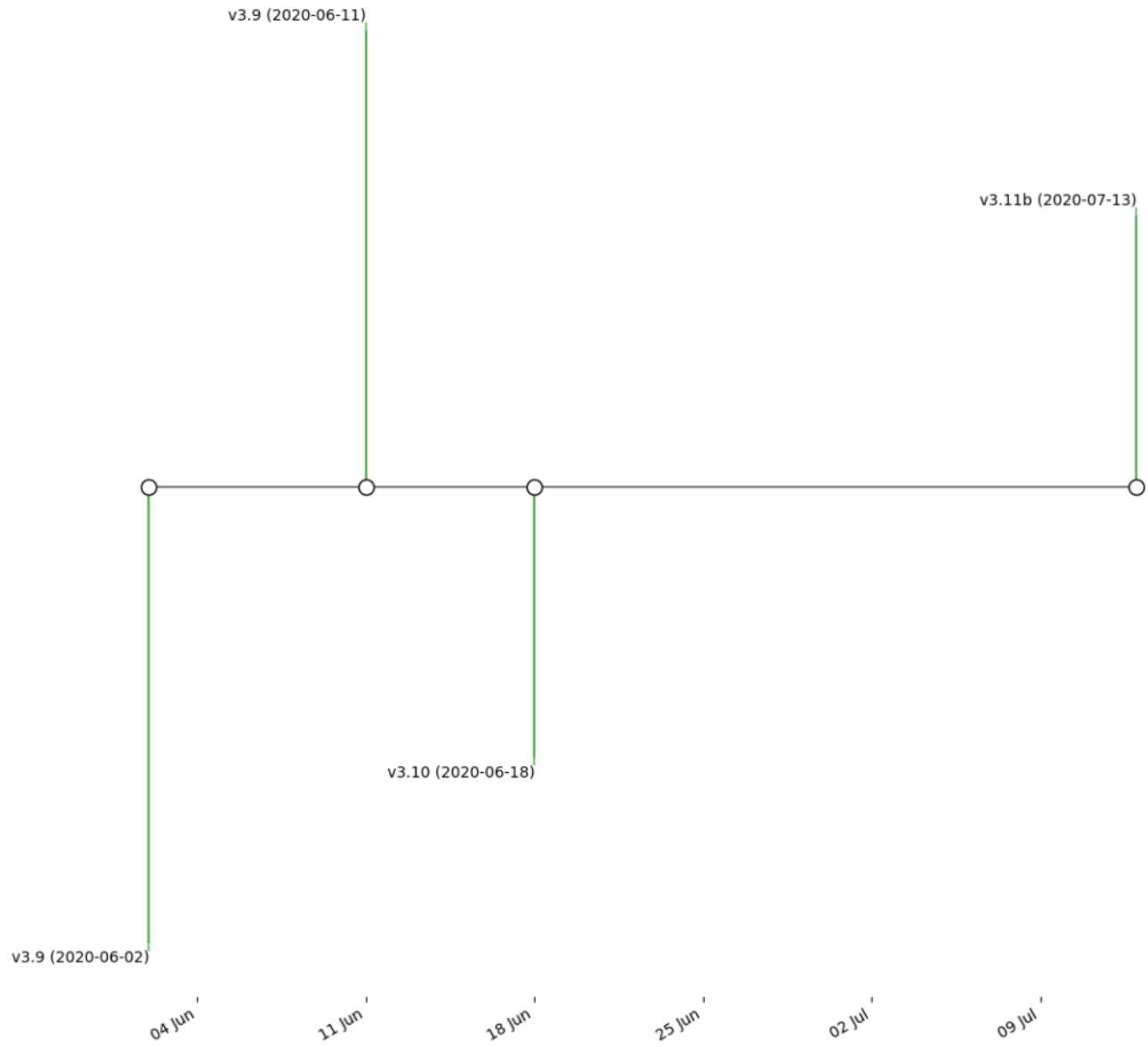
As stated by Bartholomew & Guerrero-Saade in their exceptional paper "Wave your false flags!", timestamps allow for creating a timeline of a threat actor's campaigns or deduce their working hours:

> A great benefit of the Portable Executable file format is the inclusion of compilation times. Though these can be altered with ease, many samples include original timestamps. Beyond an obvious indication of an actor's longevity, timestamps allow for an understanding of specific campaigns as well as the evolution of an actor's toolkit throughout the years. With a large enough collection of related samples, it's also possible to create a timeline of the campaign operators' workday. Where these operate in any professional setting or with any semblance of discipline, it's possible to match the normal peaks and troughs of a workday and pinpoint a general timezone for their operations.
>
> *Bartholomew & Guerrero-Saade – Wave your false flags!*

There are at least two commonly plot types possible based on the PE timestamps:

- timeline of the malware samples/families (see Page 32 of Mandiant's APT1 report for a great example)
- heatmap by day of week and hour of day (similar to the ones typically found in web traffic analysis)

Bartholomew & Guerrero-Saade stated "*With a large enough collection of related samples*" all these cool things can be done. Honestly, I don't have a large number of samples for this blog post at hand. Nevertheless, I've plotted a tiny timeline of the different SDBBot versions published (based on the PE timestamps) in Summer 2020. The samples are on MalwareBazaar.



Timeline of SDBBot versions in Summer 2020

Of course, the is room for improvement, and not speaking of the horrible design. For instance, we could plot tiny red dots on the X-axis for every day the threat actor was active. In this case, this should give us a workday / weekend pattern.

Use case: Pinpoint a threat actor to a timezone

The previous section mentions that the timezone of a threat actor can be pinpointed based on PE timestamps. In order for this to work out, there are two requirements:

- "*large enough collection of related samples*" that we can attribute to the threat actor without a doubt
- certain confidence that the threat actor did not tamper with the PE timestamps

It is likely that the PE timestamps fall into a certain range if the threat actor works in a professional environment with regular office hours. Likely, most samples will be compiled within an 8 to 12 hours range, which maps to a 9 to 5 pattern plus/minus two hours. Furthermore, there are likely one or two days off like Saturday or Sunday. But this might depend on the cultural origin of the threat actor. For instance, 996 working hour system is common in the People's Republic of China. In addition, there could also be thirty minutes and up to two hours of lunch breaks.

If you can identify such a pattern, then you add/subtract the difference of hours so that this maps to the 9 AM to 5 PM range of a timezone. In case your plot does not reveal a 9 to 5 pattern, then there are still possible explanations. In case you can see two possible 9 to 5 patterns then this could indicate shift work.

Finally, another warning example of why you should not blindly trust PE timestamps. Imagine that you've dozens of samples of a threat actor. You are pretty confident that this threat actor works normal working hours from 9 AM to 5 PM (GMT+0). You could assume that this threat actor may work in the United Kingdom or in Portugal if you blindly and solely rely on these timestamps.

Suppose that later on, you've got additional information from incident response engagements. Now, it seems that the threat actor works from 12 PM to 8 PM (GMT+0). This would map from 9 AM to 5 PM (GMT+3). Possible countries would now be, for instance, Russia or Turkey. How can you really be sure that the threat actor did not manipulate the PE timestamps and subtracted three hours from each and every timestamp found in the PE files?