Learn how to fix PE magic numbers with Malduck

0xc0decafe.com/fix-pe-magic-numbers-with-malduck/

January 28, 2021

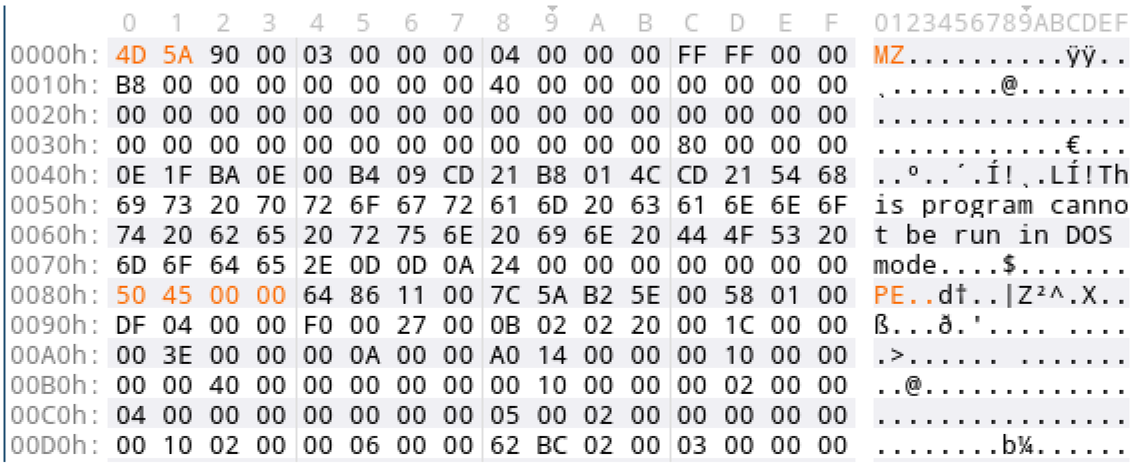
Malware often corrupts the Portable Executable (PE) header to hinder its analysis. By overwriting parts of the PE header, malware evades simple memory dumpers and thwarts proper loading by analysis tools. If we're lucky then malware only overwrites the magic numbers of the PE header (`MZ` and `PE`) and leaves the rest of the header intact. We can fix such corrupted PE headers with ease. All we need is a little bit of knowledge about the PE format and the right tool to manipulate memory dumps.

First, we'll learn how to identify such corrupted PE headers quickly with a hexeditor. Next, we'll see how to manipulate memory dumps with Malduck. This section gives you a head start on how to use this great Python module effectively. Finally, we are putting it all together and write a script to fix PE magic numbers of a corrupted PE header.

Identifying overwritten magic numbers

In the following, I assume that you've got a basic understanding of the PE format. If not, I can recommend the article on osdev.org (check the overview graphic of the PE format) as an introduction.

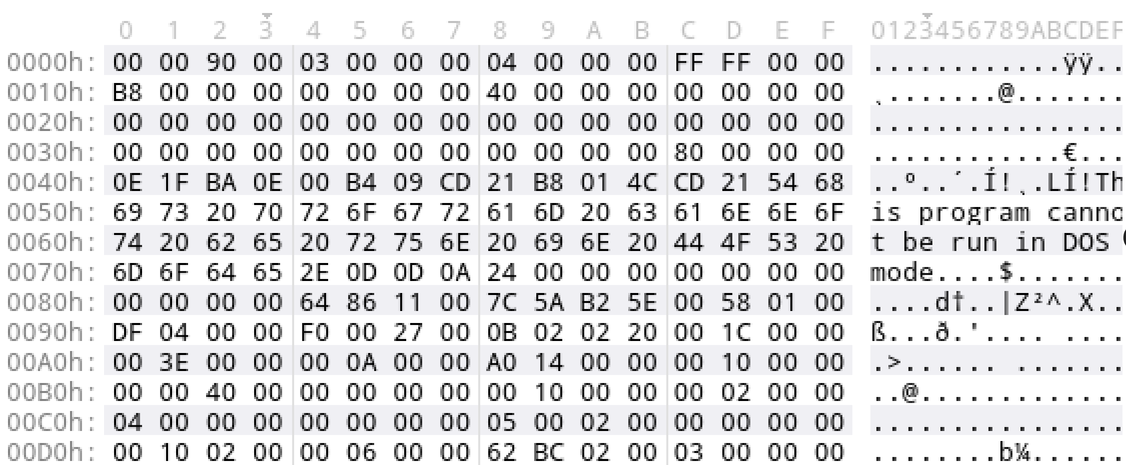
There are two magic numbers in the PE header that are frequently overwritten by malware. First, the magic number of the DOS header (`_IMAGE_DOS_HEADER`), which is a two-byte or WORD constant (`MZ`). Second, the magic number of the PE header, which is a four-byte or DWORD constant (`PE\x00\x00`). This is illustrated in the following screenshot of a PE file opened in a hexeditor. Both magic numbers are colored in orange. The `MZ` magic is at offset `0x0` and the `PE\x00\x00` is at offset `0x80` .



```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0000h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
0010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
0040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
0050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
0060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
0070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
0080h: 50 45 00 00 64 86 11 00 7C 5A B2 5E 00 58 01 00  PE..d†..|Z²^.X..
0090h: DF 04 00 00 F0 00 27 00 0B 02 02 20 00 1C 00 00  ß...ð.'.... ....
00A0h: 00 3E 00 00 00 0A 00 00 A0 14 00 00 00 10 00 00  .>...... .......
00B0h: 00 00 40 00 00 00 00 00 00 10 00 00 00 02 00 00  ..@.............
00C0h: 04 00 00 00 00 00 00 00 05 00 02 00 00 00 00 00  ................
00D0h: 00 10 02 00 00 06 00 00 62 BC 02 00 03 00 00 00  ........b¼......
```

Principal magic numbers of a PE file (orange)

Now that we've seen a perfectly fine PE header, let's see how a slightly corrupted PE header looks like. The next screenshot shows a PE header with both (principal) magic numbers overwritten.



```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0000h: 00 00 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  ............ÿÿ..
0010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
0040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
0050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
0060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
0070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
0080h: 00 00 00 00 64 86 11 00 7C 5A B2 5E 00 58 01 00  ....d†..|Z²^.X..
0090h: DF 04 00 00 F0 00 27 00 0B 02 02 20 00 1C 00 00  ß...ð.'.... ....
00A0h: 00 3E 00 00 00 0A 00 00 A0 14 00 00 00 10 00 00  .>...... .......
00B0h: 00 00 40 00 00 00 00 00 00 10 00 00 00 02 00 00  ..@.............
00C0h: 04 00 00 00 00 00 00 00 05 00 02 00 00 00 00 00  ................
00D0h: 00 10 02 00 00 06 00 00 62 BC 02 00 03 00 00 00  ........b¼......
```

Overwritten magic numbers (MZ + PE)

We can still identify that this is likely a PE file since the famous string `This program cannot be run in DOS mode` is still there. But we are missing these two magic numbers, which in turn hinders many analysis tools to properly load and analyze such a binary.

So, fixing this kind of corrupted PE header is straightforward. First, we have to restore the magic number `MZ` at offset `0x0` . Second, we have to determine the offset to the PE header. The DOS header holds this offset in its field `e_lfanew` (see next code block with DOS header for reference). Therefore, we have to read the value of `e_lfanew` from the

DOS header. `e_lfanew` resides at offset `0x3c`. Third, we have to restore the PE header magic number `PE\x00\x00` at the offset pointed to by `e_lfanew`. Finally, we should validate, if the file is now a valid PE file.

```
typedef struct _IMAGE_DOS_HEADER {
    WORD  e_magic;      /* 00: MZ Header signature */
    WORD  e_cblp;       /* 02: Bytes on last page of file */
    WORD  e_cp;         /* 04: Pages in file */
    WORD  e_crlc;       /* 06: Relocations */
    WORD  e_cparhdr;    /* 08: Size of header in paragraphs */
    WORD  e_minalloc;   /* 0a: Minimum extra paragraphs needed */
    WORD  e_maxalloc;   /* 0c: Maximum extra paragraphs needed */
    WORD  e_ss;         /* 0e: Initial (relative) SS value */
    WORD  e_sp;         /* 10: Initial SP value */
    WORD  e_csum;       /* 12: Checksum */
    WORD  e_ip;         /* 14: Initial IP value */
    WORD  e_cs;         /* 16: Initial (relative) CS value */
    WORD  e_lfarlc;     /* 18: File address of relocation table */
    WORD  e_ovno;       /* 1a: Overlay number */
    WORD  e_res[4];     /* 1c: Reserved words */
    WORD  e_oemid;      /* 24: OEM identifier (for e_oeminfo) */
    WORD  e_oeminfo;    /* 26: OEM information; e_oemid specific */
    WORD  e_res2[10];   /* 28: Reserved words */
    DWORD e_lfanew;     /* 3c: Offset to extended header */
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Note that if you encounter a memory dump that starts with around 1000 / 0x400 zero bytes, followed by properly aligned code, then you are likely out of luck. What you are looking at is likely a completely overwritten PE header. Here I've encountered two scenarios throughout the years. First, the malware unpacks a perfectly fine PE file and overwrites the PE header, for instance, before injecting it into another process. If this is the case then go back to debugging and dump the PE file before the header is overwritten.

Second, the malware unpacks a PE file with an already corrupted PE header. In this case, you have to restore the PE header. If it is really necessary then you can try to build a PE file from scratch with LIEF. However, this is out of the scope of this blog post.

Manipulating memory dumps with Malduck

Malduck is a Python module that helps writing malware analysis scripts quickly. It is developed and maintained by CERT.pl. Malduck's documentation is very decent. It is my default go-to tool to write malware analysis scripts (e.g. for aPLib decompression).

Open a memory dump with Malduck

Before we can manipulate memory dumps (of PE files), we have to open them with Malduck. The basis for all memory representations is the class `Malduck.procmem` (alias for `malduck.procmem.procmem.ProcessMemory`). The constructor takes three parameters:

- **buf**: a buffer of the memory contents (as among others Python bytes)
- **base** (optional): the base address of the memory dump, which defaults to `0x0`
- **regions** (optional): a list of regions of the memory dump, defaults to `None` (not relevant in the following since we'll work with PE dumps)

We can get the length of a `ProcessMemory` instance with the method `length` and close it with `close`.

Apart from methods for reading and writing (see next sections), there are methods for searching (**`findmz`** , `findp` , `findv` , `regexp` , `regexv`), YARA scanning (`yarap` , `yarav`), and disassembling (`disasmv`). Note that methods may be suffixed with either `v` (virtual) or `p` (physical). Methods suffixed with `v` work on virtual addresses and methods suffixed with `p` work on physical addresses (raw offsets in the memory dump). The methods `p2v` and `v2p` translate from physical addresses to virtual and vice versa.

Based on `malduck.procmem.procmem.ProcessMemory`, there are four more memory representations in Malduck:

- ProcessMemoryPE (alias `procmempe`) for PE files
- ProcessMemoryELF (alias `malduck.procmemelf`) for ELF files
- CuckooProcessMemory (alias `malduck.cuckoomem`) for memory dumps in Cuckoo 2.x format
- IDAProcessMemory (alias `malduck.idamem`) for working with IDAPython

Since this blog post covers PE files, we will work with `ProcessMemoryPE` (alias `procmempe`) in the following. The constructor differs slightly from the constructor of `ProcessMemory`. The first two parameters are still `buf` and `base`. However, it does not take the parameter `regions` but takes two more parameters:

- `image` (optional): indicate that this is a dump of a memory-mapped PE file, which defaults to `False`
- `detect_image` (optional): heuristically detects if is a memory-mapped PE file, which defaults to `False`

A useful method of `ProcessMemoryPE` is `is_valid`, which checks if the imagebase of the memory dump points to a valid PE header.

I encourage you to read the documentation to find other hidden gems. There are further methods like `extract` that tries to extract a malware configuration from the memory dump. See Malduck's static configuration extractor engine for more information.

Read ProcessMemoryPE

Malduck supports two ways to read from a `ProcessMemory` instance. First, it allows reading raw data chunks with `readp`, `readv`, and `readv_until`. While `readp` takes as input a raw `offset` and an optional `length`, `readv` takes as input a virtual `addr`. The method `readv_until` is useful when you want to read until a certain stop marker (e.g. end of configuration).

Second, Malduck supports reading various data types:

- strings: `asciiz` and `utf16z`
- signed integers: `int8p` / `int8v`, `int16p` / `int16v`, `int32p` / `int32v`, `int64p` / `int64v`
- unsigned integers: `uint8p` / `uint8v`, `uint16p` / `uint16v`, `uint32p` / `uint32v`, `uint64p` / `uint64v`

Note that there is always a physical (`p`) and virtual (`v`) version. Internally, all utilize either `readp` or `readv` to read the data.

Write ProcessMemoryPE

The support for writing `ProcessMemory` instances is rudimentary when compared with the reading support. There are just two functions to know: `patchp` and `patchv`. Both accept a raw `offset` / virtual `addr` and a bytes `buf`. That's it, pretty straight forward!

Putting it all together: fix PE magic numbers with Malduck

This section puts it all together: our theoretical knowledge about the PE format and our practical knowledge about memory dump manipulation with Malduck. The script `fix_pe_magic_numbers.py` takes a path to a dump of PE file with a corrupted header as input and outputs a fixed dump.

First, it loads the dump into a buffer `data` and opens it with `malduck.procmempe` (alias of `malduck.procmem.procmempe.ProcessMemoryPE`) in line 11. The method `is_valid` checks if a `ProcessMemoryPE` object is a valid PE file (line 13). Next, it patches the `MZ` magic number with the method `patchp` (line 17) and reads the DOS header field `e_lfanew` with the method `uint32p`. Again, `e_lfanew` resides at offset `0x3C`. Afterwards, it patches the `PE\x00\x00` magic number with the method `patchp` (line 24). Finally, it validates the PE file with the method `is_valid` (line 26). If it is valid, then it writes all bytes of the `ProcessMemoryPE` object to a file (line 29).

```
import sys
import malduck

def main(argv):
   if len(argv) != 2:
   print('Usage: fix_pe_magic_numbers.py PATH_TO_DUMP')
        return

    with open(argv[1], 'rb') as f:
        data = f.read()
        pe = malduck.procmempe(buf=data)

        if pe.is_valid():
            print('This file is already a valid PE file. Skipping...')
            return

        pe.patchp(0, b'MZ')
        lfanew = pe.uint32p(0x3c)
        if lfanew > len(data):
            print('Bogus lfanew value ({hex(lfanew)}). Bailing out...')
            return

        print(f'lfanew: {hex(lfanew)}')
        pe.patchp(lfanew, b'PE\x00\x00')

        if pe.is_valid():
            print('Fixed file successfully. Dumping to new file...')
            with open(argv[1].replace('bin', '') + '_fixed_header.bin', 'wb') as g:
         g.write(pe.readp(0))
                print('Done.')
        else:
            print('Could not fix file, still not a valid PE file.')

        pe.close()

if name == 'main':
    main(sys.argv)
```

Let’s see how it works with our broken memory dump from the beginning:

```
> file memdump.bin
memdump.bin: data
> python fix_pe_magic_numbers.py memdump.bin
 lfanew: 0x80
 Fixed file successfully. Dumping to new file…
 Done.
> file memdump_fixed_header.bin
memdump_fixed_header.bin: PE32+ executable (console) x86-64, for MS Windows
```

et voilà! The script fixed the memory dump as we can see in the following screenshot:

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
0010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
0040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
0050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
0060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
0070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
0080h: 50 45 00 00 64 86 11 00 7C 5A B2 5E 00 58 01 00  PE..d†..|Z²^.X..
0090h: DF 04 00 00 F0 00 27 00 0B 02 02 20 00 1C 00 00  ß...ð.'.... ....
00A0h: 00 3E 00 00 00 0A 00 00 A0 14 00 00 00 10 00 00  .>...... .......
00B0h: 00 00 40 00 00 00 00 00 00 10 00 00 00 02 00 00  ..@.............
00C0h: 04 00 00 00 00 00 00 00 05 00 02 00 00 00 00 00  ................
00D0h: 00 10 02 00 00 06 00 00 62 BC 02 00 03 00 00 00  ........b¼......
```

Fixed PE header with MZ and PE magic numbers restored

Both magic numbers are located at their correct offsets: the magic number of the DOS header `MZ` resides at offset zero and the magic number of the File header resides at offset 0x80 (as indicated by `e_lfanew`). Now you can load the PE file with other analysis tools and continue your analysis.