6845 - Extended Project

Project: Replacing DOS boot code of the MBR

Group 34:

- Matthew Ta (z5061797)
- Jordan Isakka (z5165167)
- Minh Thien Nhat Nguyen (z5137455)

Table of Contents

Background/Preparation

In order to begin our project, we needed to develop a strong understanding of the bootstrapping process, communication with the BIOS and how to we can overwrite the MBR in an effective manner.

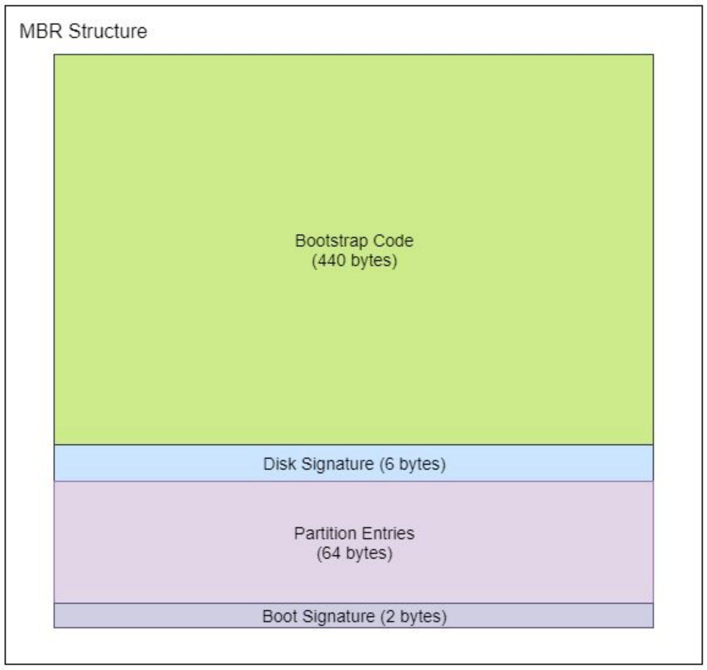
The MBR is 512 bytes in size (one sector) and consists of executable code followed by a partition table. The executable code is written in 16 bit assembly which is also known as “real mode” and is responsible for booting up the operating system by scanning the partition table to look for a bootable entry.

Once a bootable entry is found, the entry is parsed and the starting LBA of the partition is noted. The next stage of the bootloader would load the VBR of the bootable partition in one of two methods:

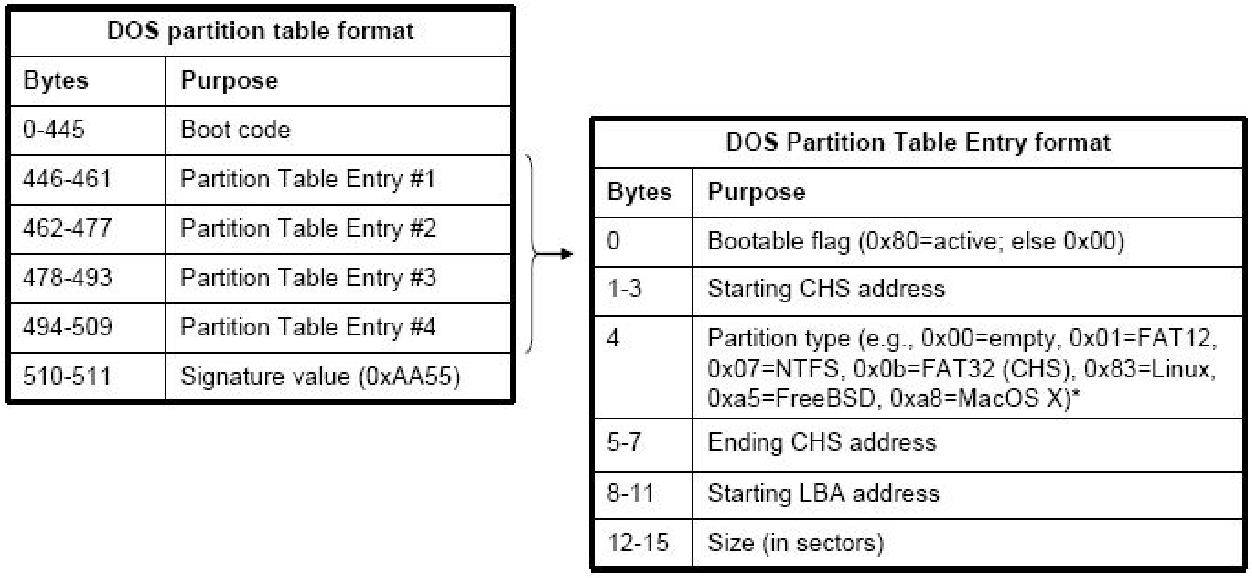
CHS- The VBR is located by the cylinder, head and sector.
LBA- The VBR is located by its absolute sector number (from sector 0).

There are fields in the partition entry that cater to both methods. Both of these methods require using interrupts from that are provided by the BIOS which allow us to read/write sectors to and from disk. Once the VBR is loaded into physical memory, the bootloader (MBR) will jump to this location and execute the code in the VBR which would ultimately boot the operating system.

Structure of the MBR



Structure of each partition entry:



DOS partition table format	
Bytes	Purpose
0-445	Boot code
446-461	Partition Table Entry #1
462-477	Partition Table Entry #2
478-493	Partition Table Entry #3
494-509	Partition Table Entry #4
510-511	Signature value (0xAA55)

DOS Partition Table Entry format	
Bytes	Purpose
0	Bootable flag (0x80=active; else 0x00)
1-3	Starting CHS address
4	Partition type (e.g., 0x00=empty, 0x01=FAT12, 0x07=NTFS, 0x0b=FAT32 (CHS), 0x83=Linux, 0xa5=FreeBSD, 0xa8=MacOS X)*
5-7	Ending CHS address
8-11	Starting LBA address
12-15	Size (in sectors)

Hardcore Technical Details

- Important BIOS interrupts:

Interrupt code	Arguments:	Description:
0x10 %ah = 0xe	%al = the character to print	Print a character to the screen
0x13 %ah = 0x2 or %ah = 0x3	%al = number of sectors %bx = buffer %dh = head number %dl = drive number	Read or write to a sector
0x13 %ah = 0x42 or %ah = 0x43	%dl = drive number %si = address packet	Extended read/write
0x16 %ax = 0x0	None	Reads a keystroke and returns its ASCII char in %al.

The tools we used in this project

Nasm/Ndisasm - For compilation of the boot code and decompilation of existing flash drives.
Qemu - A hardware emulator used to run the disk images we tampered with.
Gdb - For debugging and walking through the boot code.
Bochs - Another debugger specifically made for 16 bit assembly.
IDA - Disassembling the master boot record from various hard drives.
HxD - Hex editor that allows us to open and edit a physical drive.
VMware - Observe the effects of our malicious boot loader code in a controlled environment.

Showcase

Hiding partitions

The first thing we tried was hiding a partition entry from the BIOS.
We could achieve this by:

1. Copying the MBR into memory
2. Overwriting the partition table entry in the copy
3. Writing the copy of the MBR back onto the disk

When run, the entry would be zeroed-out and therefore not detected as a bootable partition.

Because the MBR is static in terms of structure and location, we could overwrite any partition entry by simply changing the offset.

```
%define PARTITION_OFFSET        0x1be
%define PARTITION_ENTRY_SIZE    0x10
```

In the case above, the second partition would be hidden (partitions entries start after the 446th byte, plus 16 to skip the first entry = 462 = 0x1be).

We could also use this method to overwrite specific fields in a partition entry, such as the number of sectors a partition has.

A potential application of partition hiding might be the concealment of a rootkit, or any other arbitrary data that would want to be hidden from a user.

Encrypting Partitions

Using a similar method to the hiding of a partition, we may also encrypt it:

1. Copy the MBR into memory
2. XOR a partition entry with a given key
3. Write the copy of the MBR to disk

Encrypting will allow us to retain the partition address, but still make it unreadable to the BIOS. This sort of method may be present in ransomware, decryption only occurring if the user complies with the attacker’s demands.

Ransomware

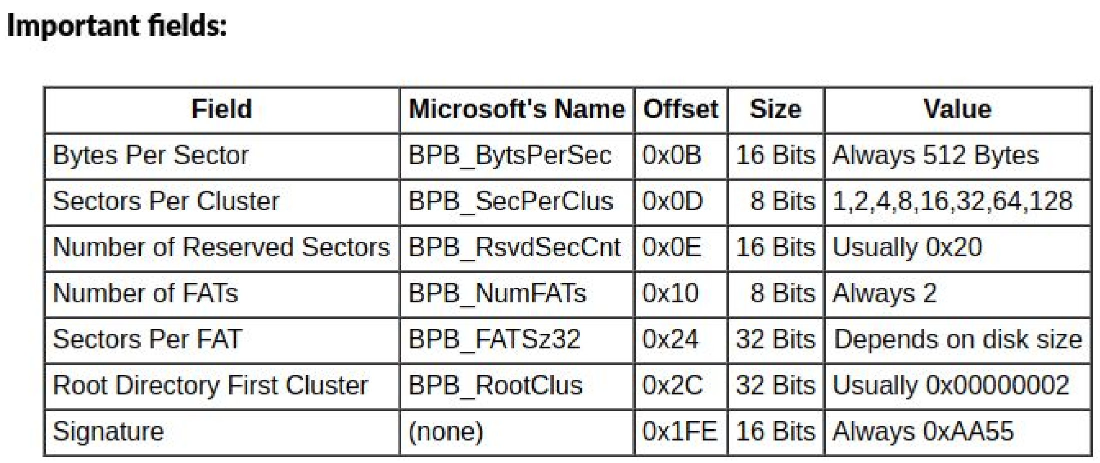
We then tried to read input from keyboard to have password checking when booting the machine.

1. Inside the original MBR, we had a jump to another sector (sector 2) for input reading and checking, and then return back to continue booting the OS
2. In another sector, we tried to check input from keyboard by reading and comparing each key pressed with the corresponding character in the key.
3. If the input didn't match then we would try to encrypt the VBR (which is used to boot the OS for bootable partition) so that the system would not boot and run

By doing this, we can make a simple ransomware by asking for password, and if it fails the check, we can halt or even encrypt data or the mbr.

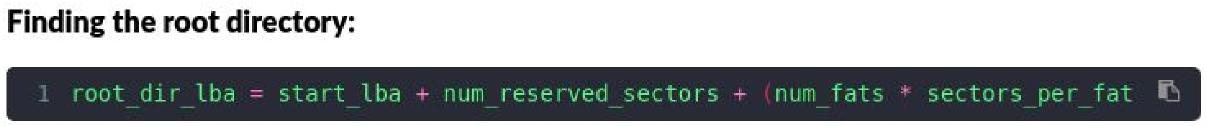
File persistence for FAT32 filesystem

For this part of the project, we attempted to develop some real mode assembly code to inject a file directly into the FAT32 filesystem. To be able to inject a file into the FAT32 file system, the structure of the FAT32 file system must first be understood. Below, the important parts of the FAT32 file system are illustrated:



Important fields:

Field	Microsoft's Name	Offset	Size	Value
Bytes Per Sector	BPB_BytsPerSec	0x0B	16 Bits	Always 512 Bytes
Sectors Per Cluster	BPB_SecPerClus	0x0D	8 Bits	1,2,4,8,16,32,64,128
Number of Reserved Sectors	BPB_RsvdSecCnt	0x0E	16 Bits	Usually 0x20
Number of FATs	BPB_NumFATs	0x10	8 Bits	Always 2
Sectors Per FAT	BPB_FATSz32	0x24	32 Bits	Depends on disk size
Root Directory First Cluster	BPB_RootClus	0x2C	32 Bits	Usually 0x00000002
Signature	(none)	0x1FE	16 Bits	Always 0xAA55



Finding the root directory:

```
root_dir_lba = start_lba + num_reserved_sectors + (num_fats * sectors_per_fat
```

The assembly code starts out like a normal MBR except, instead of looking for a bootable partition, we are seeking a FAT32 partition. If the type field in the partition table entry of the MBR is equal to 0xb or 0xc then we have a found a FAT32 partition and we can then load the VBR into memory.

The fields above can be found in the VBR belonging to the FAT32 partition. These fields can be used to determine the sector number of the root directory in LBA units. Once the location of the root directory is known, our assembly code simply loads one sector of the root directory into memory and then we simply insert an entry before writing the sector back to disk. The structure of a root directory entry is shown below:

Find the LBA of an entry:

- **Directory entry structure:**

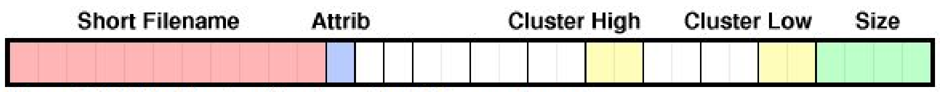


Figure 6: 32 Byte Directory Structure, Short Filename Format

Field	Microsoft's Name	Offset	Size
Short Filename	DIR_Name	0x00	11 Bytes
Attrib Byte	DIR_Attr	0x0B	8 Bits
First Cluster High	DIR_FstClusHI	0x14	16 Bits
First Cluster Low	DIR_FstClusLO	0x1A	16 Bits
File Size	DIR_FileSize	0x1C	32 Bits

```
cluster_lba = root_dir_lab + (cluster_num - 2) * sectors_per_cluster
cluster_num = cluster_hi | cluster_lo
```

After inserting the entry into the root directory, we need to also mark the entry in the FAT table at the index denoted by the cluster number in the new root directory entry as 0xFFFFFFF0 which means end of chain. If we didn’t do this then the FAT32 entry will contain 0x00000000 which essentially means free cluster. This would result in file system errors and we would not be able to open the file that we injected.

To insert an entry into the FAT table, the assembly code first reads the FAT table from disk into memory using the bio interrupts and then adds the entry in memory before copying the FAT table back to the appropriate sector on the disk. This needs to be done twice since there are two FAT tables in a FAT32 file system.

In order to install this on the system, our custom mbr code (custom_mbr.asm) needs to be placed at sector 0 overwriting the original MBR. This code will be placed at sector 1 and the data to inject will be placed in the sector after.

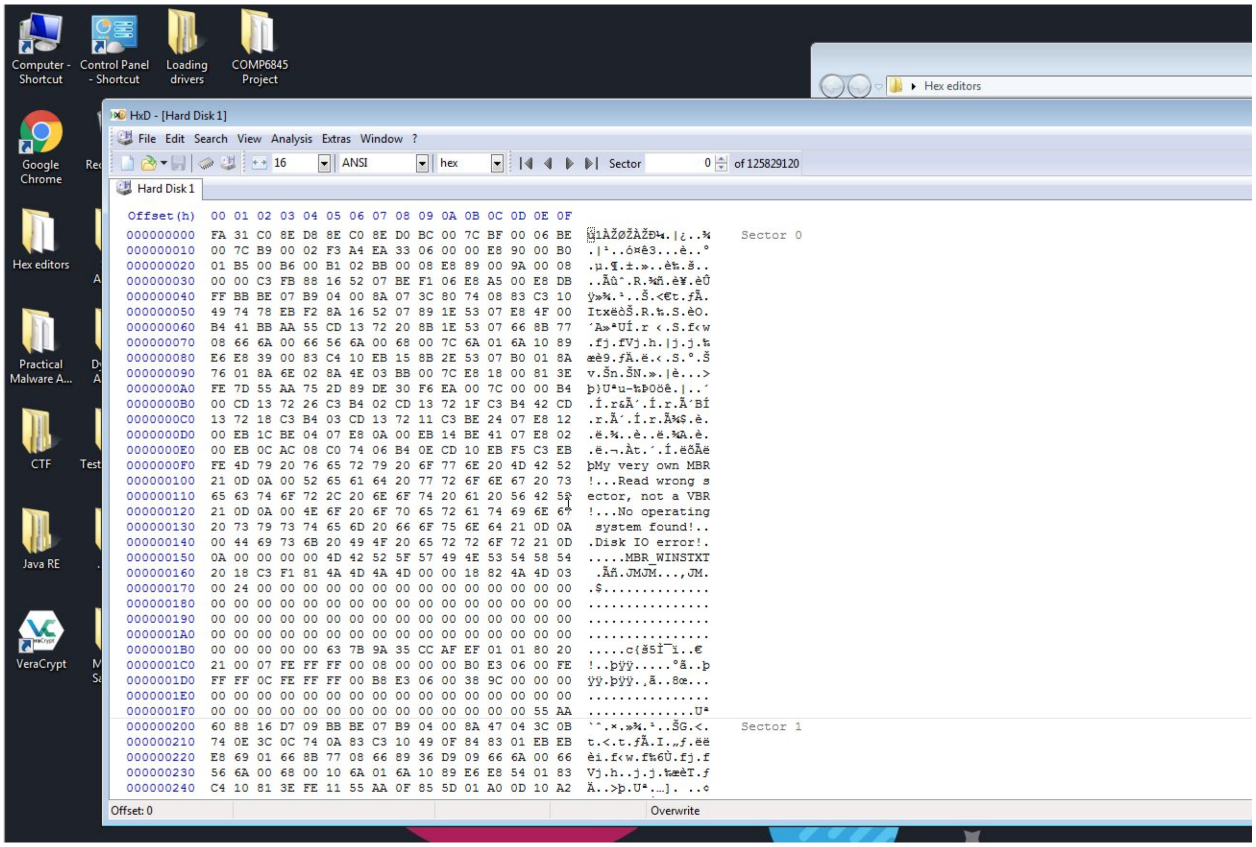
The implication of this method is that removal of malware that achieves persistence in this manner is extremely difficult and the forensic investigator would need to be able to understand 16 bit assembly code to be able to reverse engineer the MBR in order to find the malware and remove it. Bootkits generally use this method to achieve persistence in an infected host.

Note: All reading and writing of sectors was done in LBA mode since the FAT32 disk that we were working on resides past 8GB so CHS was out of the question.

Method of testing

To test our malicious master boot records, a Windows 7 virtual machine was created in VMware. A snapshot was first created so if we are not able to boot back into Windows 7, we can simply restore the virtual machine to a clean state. A FAT32 partition was also created to demonstrate the file persistence code.

After compiling our code with nasm, we used HxD to open the compiled code as well as our physical drive and overwrite the MBR with our code in sector 0 (which would load and run code in other sector, e.g. sector 1). Sector 1 will contain our payload i.e the code we want to run. Sector 2 will usually contain any extra data we need (used for file persistence).



```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

000000000  FA 31 C0 8E D8 8E C0 8E D0 BC 00 7C BF 00 06 BE  ú1ÀŽØŽÀŽÐ¼.|¿..¾     Sector 0
000000010  00 7C B9 00 02 F3 A4 EA 33 06 00 00 E8 90 00 B0  .|¹..ó¤ê3...è..°
000000020  01 B5 00 B6 00 B1 02 BB 00 08 E8 89 00 9A 00 08  .µ.¶.±.»..è‰.š..
000000030  00 00 C3 FB 88 16 52 07 BE F1 06 E8 A5 00 E8 DB  ..Ãû^.R.¾ñ.è¥.èÛ
000000040  FF BB BE 07 B9 04 00 8A 07 3C 80 74 08 83 C3 10  ÿ»¾.¹..Š.<€t.ƒÃ.
000000050  49 74 78 EB F2 8A 16 52 07 89 1E 53 07 E8 4F 00  ItxëòŠ.R.‰.S.èO.
000000060  B4 41 BB AA 55 CD 13 72 20 8B 1E 53 07 66 8B 77  ´A»ªUÍ.r ‹.S.f‹w
000000070  08 66 6A 00 66 56 6A 00 68 00 7C 6A 01 6A 10 89  .fj.fVj.h.|j.j.‰
000000080  E6 E8 39 00 83 C4 10 EB 15 8B 2E 53 07 B0 01 8A  æè9.ƒÄ.ë.‹.S.°.Š
000000090  76 01 8A 6E 02 8A 4E 03 BB 00 7C E8 18 00 81 3E  v.Šn.ŠN.».|è...>
0000000A0  FE 7D 55 AA 75 2D 89 DE 30 F6 EA 00 7C 00 00 B4  þ}Uªu-‰Þ0öê.|..´
0000000B0  00 CD 13 72 26 C3 B4 02 CD 13 72 1F C3 B4 42 CD  .Í.r&Ã´.Í.r.Ã´BÍ
0000000C0  13 72 18 C3 B4 03 CD 13 72 11 C3 BE 24 07 E8 12  .r.Ã´.Í.r.Ã¾$.è.
0000000D0  00 EB 1C BE 04 07 E8 0A 00 EB 14 BE 41 07 E8 02  .ë.¾..è..ë.¾A.è.
0000000E0  00 EB 0C AC 08 C0 74 06 B4 0E CD 10 EB F5 C3 EB  .ë.¬.Àt.´.Í.ëõÃë
0000000F0  FE 4D 79 20 76 65 72 79 20 6F 77 6E 20 4D 42 52  þMy very own MBR
000000100  21 0D 0A 00 52 65 61 64 20 77 72 6F 6E 67 20 73  !...Read wrong s
000000110  65 63 74 6F 72 2C 20 6E 6F 74 20 61 20 56 42 52  ector, not a VBR
000000120  21 0D 0A 00 4E 6F 20 6F 70 65 72 61 74 69 6E 67  !...No operating
000000130  20 73 79 73 74 65 6D 20 66 6F 75 6E 64 21 0D 0A   system found!..
000000140  00 44 69 73 6B 20 49 4F 20 65 72 72 6F 72 21 0D  .Disk IO error!.
000000150  0A 00 00 00 00 4D 42 52 5F 57 49 4E 53 54 58 54  .....MBR_WINSTXT
000000160  20 18 C3 F1 81 4A 4D 4A 4D 00 00 18 82 4A 4D 03   .Ãñ.JMJM...,JM.
000000170  00 24 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .$..............
000000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000001B0  00 00 00 00 00 63 7B 9A 35 CC AF EF 01 01 80 20  .....c{š5Ì¯ï..€ 
0000001C0  21 00 07 FE FF FF 00 08 00 00 00 B0 E3 06 00 FE  !..þÿÿ.....°ã..þ
0000001D0  FF FF 0C FE FF FF 00 B8 E3 06 00 38 9C 00 00 00  ÿÿ.þÿÿ.¸ã..8œ...
0000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............Uª
000000200  60 88 16 D7 09 BB BE 07 B9 04 00 8A 47 04 3C 0B  `ˆ.×.»¾.¹..ŠG.<.     Sector 1
000000210  74 0E 3C 0C 74 0A 83 C3 10 49 0F 84 83 01 EB EB  t.<.t.ƒÃ.I.„ƒ.ëë
000000220  E8 69 01 66 8B 77 08 66 89 36 D9 09 66 6A 00 66  èi.f‹w.f‰6Ù.fj.f
000000230  56 6A 00 68 00 10 6A 01 6A 10 89 E6 E8 54 01 83  Vj.h..j.j.‰æèT.ƒ
000000240  C4 10 81 3E FE 11 55 AA 0F 85 5D 01 A0 0D 10 A2  Ä..>þ.Uª.…]. ..¢
```

We then restart the Windows virtual machine to let our code execute. Once windows has booted, we go into file explorer to see the effects of our code.

Problems Encountered

- Debugging QEMU with GDB is sometimes difficult since GDB interprets the bootloader instructions in 32 bit mode instead of 16 bit mode. This makes is tedious to step through the code.
- The first time we overwrite the original MBR with our MBR, we forgot to include the disk signature which prevented us from booting Windows since the Windows boot manager would issue an error indicating it could not locate the disk.
- While trying to encrypt the partitions, the windows boot manager would detect the partition as invalid and would automatically clean up the partition, restoring it to its correct state.

Acronym Glossary

BIOS: Basic Input/Output System
CHS: Cylinder-Head-Sector
FAT: File Allocation Table
LBA: Logical Block Addressing
MBR: Master Boot Record
OS: Operating System
VBR: Volume Boot Record

Reference Materials

http://www.ctyme.com/intr/int.htm
https://www.pjrc.com/tech/8051/ide/fat32.html
https://thestarman.pcministry.com/asm/mbr/PartTables.htm
https://blog.ghaiklor.com/how-to-implement-your-own-hello-world-boot-loader-c0210ef5e74b
http://samuelkerr.com/?p=262
https://countuponsecurity.com/2017/07/02/analysis-of-a-master-boot-record-eternalpetya/
https://phocean.net/2011/01/20/debugging-the-mbr-with-ida-pro-and-bochs.html
https://stackoverflow.com/questions/14242958/debugging-bootloader-with-gdb-in-qemu
https://blog.ghaiklor.com/how-to-implement-your-own-hello-world-boot-loader-c0210ef5e74b