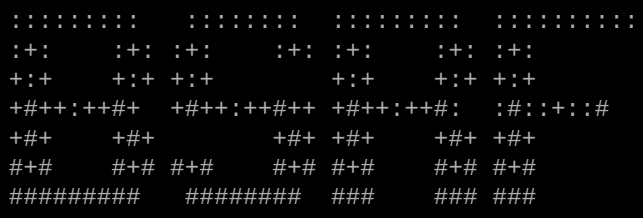
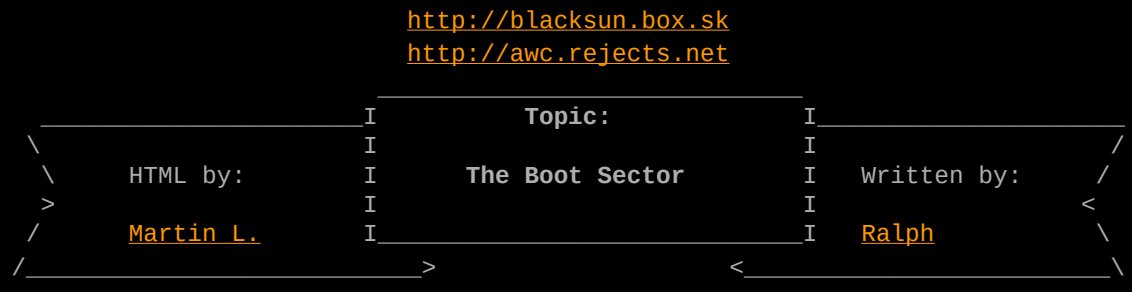


http://blacksun.box.sk
http://awc.rejects.net



Version: 1.0 Date: 7/31/00

TOC

1. Introduction

Well usually I give you a specific purpose at this point, but in this case I can't. I was just in the mood to write something on boot sectors so I did it. And maybe (hopefully) someone out there can make use of this info. This thing will most likely became part of a larger tutorial, maybe something on assembly or on OS design. After having consumed this text file you should know enough to design and create your own boot sector, maybe for a virus, or an OS, or...?

What you need

Before reading this you should have a basic knowledge of assembly. If you don't, read my other tutorial, it's called Sk00l m3 ASM!!#@$!@# and is available from awc.rejects.net We will be using 2 different programs to code this shit: NASM and TASM. NASM is freely available from http://www.web-sites.co.uk/nasm/, but TASM you have to buy. I don't like piracy, but if you're just gonna use TASM this once, don't bother spending $150 on it. There are plenty of sites that have a copy.

Why am I using 2 different programs? Well I have always used NASM to make simple programs as it's good at creating efficient memory copies. I always use TASM to make programs a bit more complex. In the end however it comes down to the answer "why not??". However, it shouldn't be hard at all to make the TASM program in NASM (or the other way around), just change a few things here and there. If enough people come bitch to me, I'll rewrite all the code for NASM/TASM.

2. Basic hard drive/BIOS shit

As soon as you flip that switch, your CPU starts executing shit located at F000:FFF0. This area contains the BIOS, Basic Input/Output System. This code is written in assembly and is stored in chips called EPROMs in your computer. This code will perform something known as POST, Power On Self Test. This checks for installed devices

and checks if they all work. In particular it checks for the video card and runs the video BIOS usually located at C000h. Next it checks for other ROMs to see if they have installed BIOSes. Usually it then finds and executes the hard drive BIOS located at C8000h. Then it starts something like a "system inventory" where it checks for other installed devices and tests them. It does some more stuff that's all basicly useless for us right now, until it finally transfers control over to the operating system. That's the part that we're interested in. Back in the old days, only one OS was installed on a computer. If you bought a certain computer, you could only run the OS that was made for it. Nothing else. Obviosly that wasn't such a good thing as you would have to buy a new computer if you wanted a different OS, so BIOS makers came up with the Boot Sector. In case you didn't know yet, a Sector is the smallest area your hard drive can access. According to the ATA standards each sector is exactly 512 bytes. However ATA standards only apply to hard drives, things like floopies can use whatever they want. Knowing this we can move on to the boot sector.

3. Making a Boot Sector

After the BIOS has successfully completed the POST it calls interrupt 19h. You can actually see this by dumping the memory located at F000:FFF0. For example, on my box I used debug with the following result:
-d f000:fff0
F000:FFF0 CD 19 E0 00 F0 31 31 2F-32 36 2F 39 39 00 FC 81 .....11/26/99...

As you should know, CD = INT. INT 19h attempts to read in the Boot Sector of the 1st floppy disk. If it fails it does the same thing on the 1st hard drive. If that fails it returns an error message. A valid boot sector must have its last two bytes set to AA55h. Assuming a valid boot sector is found, the code is loaded into memory at location 0000:7C00 and interrupt 19h jumps there to start executing the code. Since a boot sector has to fit into one sector (512 bytes) it can't really do much, usually it does a search for another file on another sector, then executes it. Our boot sector won't do that. For now it is enough that it displays a message and reboots when you press a key. Since DOS is not loaded yet, we have to use BIOS interupts to do all this. First we display a messages using interupt 10h. Next we wait for the user to press a key using interrupt 16h, and finally we make a FAR jump to FFFF:0000 which we restart the computer. So lets code this bitch:

First we use the code

```
MOV AX,0x0003
INT 0x10
```

to get into video mode. The registers have to be set up like this:

AH Function number (00h, video)
AL Video Mode (03, 80x25x16)

Next we print the message using:

```
MOV AX,0x1301
MOV BX,0x0007
MOV CX,0x23
MOV BP,MSG
ADD BP,0x7C00
INT 0x10
```

AH Function number (13h: print string)
AL Write Mode (01h: string is characters only, attribute in BL, cursor moved)
BH Video Page number (00h)
BL Attributes of characters (07h)
CX Length of string, excluding any attributes (23h = 35 characters)
BP ES:BP must point to the string, since a boot sector starts at 07C00, we add that to BP after we loaded it. You could also set the entry point of the program to 07C00, or change the data segment register to point to 07C00, but since it's just one instruction, this is fine for now.

Now we wait for the key to be pressed:

```
MOV AH,0x00
INT 0x16
```

Registers:
AH - 00, Read keyboard buffer, wait till full if not already.
The buffer will be empty since the computer didn't get time to put anything into it yet. Finally we reboot the computer by simply jumping to 0000:FFFF:

```
        DB 0xEA
        DW 0x0000
        DW 0xFFFF
```

This looks a bit wierd but it's actualy quite simple. When declaring "variables" in assembly, the assembler simply puts the value into a memory location. Usually you use interrupts or something to point to them in order to use and manipulate them, but we could also put code there. This is what we're doing here. If you get a Hex to Mnemonix chart you will notice that EA is a Far Jump. So we put that into memory, followed by the location to jump to.

Next we fill the the remaining memory with NULL:

```
        TIMES 510-($-$$) DB 0
```

This could also be done in TASM with something like TIMES 510 DUP (0). Finally we have to add those two bytes to the end so that the BIOS will know that this is a valid boot sector. This is done with the simple statement:

```
        SIGNATURE DW 0xAA55
```

Here is the full code to everything we just discussed:

```
START:
        MOV AX,0x0003
        INT 0x10
PRINT_STRING:
        MOV AX,0x1301
        MOV BX,0x0007
        MOV CX,0x23
        MOV BP,MSG
        ADD BP,0x7C00
        INT 0x10
WAIT_FOR_KEY_PRESS:
        MOV AH,0x00
        INT 0x16
REBOOT:
        DB 0xEA
        DW 0x0000
        DW 0xFFFF
MSG DB 'pR3sS 4nY k3y 2 k0n71nu3',13,10,'btw, ph33r',0
        TIMES 510-($-$$) DB 0
SIGNATURE DW 0xAA55
```

Assemble with "nasm filename.asm". This will get you a file called "filename", no extension. It is a raw binary image of the code. Get out a floppy and type "debug filename". Enter this at the prompt: w 100 0 0 1. You should know what this does from my assembly tutorial, if not it simply means write whatever is in memory to location 100 on disk 0 (A:), starting from sector 0 to sector 1. Now try booting from this disk. You should get the message:

```
        pR3sS 4nY k3y 2 k0n71nu3
        btw, ph33r
```

And when you press a key, the keyboard buffer gets filled so interupt 16h is finished and we move on to the restart procedure. Obviously this was just a simply example, instead of printing a string, waiting for a key press and restarting, you could've put anything in there, just as long as you don't use DOS interupts. One nice thing might be to get into Protected Mode, or you could even do some graphics shit which might run faster than in DOS or Windows since nothing is in memory except what you want to be there.

4. Making a program to write a boot sector

If you tryed to access the disk with your boot sector on it, you'll notice that you can't. At least not using DOS. That's because DOS uses a few bytes of memory for data that it needs to know in order to determine what kind of disk it is, our program however uses those bytes for the code. Now, you could look up those memory areas and declare them at the start of your program, but instead we will just create a program that will write any kind of file directly to the boot sector of a disk, regardless of what's on that disk. This sounds harder than it really is. In fact, the resulting program is a mere 73 bytes. First of all we have to open the file we want to write to the boot sector using the code:

```
READFILE:

        MOV AX,3D00h
        MOV DX,OFFSET FILENAME
        INT 21h
```

AH = 3Dh, Open file
AL = 00, open file as read only
DX = Points to file name. This has to be a ASCIIZ string, meaning it's terminated with a NULL character (0).

This will return the file handle in AX. If an error has occured, the carry flag will be set and the error code stored in AH. In that case, branch:

```
        JC ERROR
```

Otherwise proceed to reading in the file:

```
        MOV BX,AX
        MOV AH,3Fh
        MOV CX,0200h
        MOV DX,OFFSET SHIT
        INT 21h
```

First we move the file handle from AX into BX, then set up the other registers as follows:
AH = 3Fh, Read file
CX = 200h, Amount of data to read. Since a boot sector will always be 512 bytes long we read in 200h bytes (512d).
DX = Points to memory area to hold contents of file
Again, the carry flag will be set if an error occured, so branch:

```
        JC ERROR
```

Now we're getting to the actual writing part. First we reset the floppy disk controller with the code:

```
WRITE_SECTOR:

        MOV AH,0h
        MOV DL,0
        INT 13h
```

Next we write the data:

```
        MOV AX,0301h
        MOV CX,1
        MOV DX,0
        MOV BX,OFFSET SHIT
        INT 13h
```

This is one of the more complicated interupts, and you have to know some shit about how hard drives are made up.
AH = 03h, Write Sector
AL = 1, Number of sectors to write on same track and head
CH = 0, Track number to write
CL = 1, Sector number to start writing from
DH = 0, Head number to write

DL = 0, Drive number to write (0 = A, 1 = B, etc)
BX = Buffer to write sector(s) from

Again the carry flag is set if an error occurs, but I like to keep things interesting and used a different method to check for an error. The error code is stored in AH, if AH is 0 there was no error. So to check for an error I can simply XOR AH, AH and Jump if Not Zero.

```
        XOR AH,AH
        JNZ ERROR
```

Otherwise, we're done and can terminate the program:

```
        INT 20h
```

So the finished program looks like this:

```
MAIN SEGMENT                    ;the usual setup I use for .com files

        ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN
        ORG 100h

START:

READFILE:                       ;reads file as explained above

        MOV AX,3D00h
        MOV DX,OFFSET FILENAME
        INT 21h

        JC ERROR

        MOV BX,AX
        MOV AH,3Fh
        MOV CX,0200h
        MOV DX,OFFSET SHIT
        INT 21h

        JC ERROR

WRITE_SECTOR:                   ;writes sectors as explained above

        MOV AH,0h
        MOV DL,0
        INT 13h

        MOV AH,03h
        MOV AL,1
        MOV CX,1
        MOV DX,0
        MOV BX,OFFSET SHIT
        INT 13h

        XOR AH,AH
        JNZ ERROR

        INT 20h

ERROR:

MOV AH,09h                      ;displays error message

        MOV DX, OFFSET SHIT1
```

```
        INT 21h
        INT 20h

SHIT      DB ?          ;uninitialized array to hold contents of file
SHIT1     DB 'Error$'   ;Bad ass error message
FILENAME DB 'ph33r',0 ;filename to write
MAIN ENDS
END START
```

Now this thing is very very basic. There are many areas you could improve on. For example:

1. Make the filename a user inputed value. To do so, make FILENAME an array of 12 unitialized bytes (DOS filenames can't be longer than that). Than load that array into SI and call interupt 16h, function 0h. Loop it until enter is pressed, store the value in SI, incrementing SI each time.
2. Add more error messages, maybe even something that checks the error code and response with an appropriate message
3. This program wont wait for the motor to start up, so make a loop that loops about 3 times, checking if the disk drive is ready. If all tries fail, return an error saying that the disk is not in the drive or something. The error code is returned in AH, so you can make a simple check and respond with the corrosponding error message.
4. Display a (C) Microsoft message

5. Other

If you fuck up your computer as a result of this tutorial, don't blame me. All code has been tested and works great, but I cannot be held responsible for anything that happens to you as a result of using this information.

You may freely distribute this text as long as you don't change anything. If there's something you think should be changed, contact me first.

Please always get the newest version of this an other tutorials at http://awc.rejects.net as they usually contained updated information, and addons.

Send feedback to fu@ckz.org

Greetings to:
cozgedal, skin_dot, Linxor, jyc, rpc, moJoe, Lindex, aphex twin

```
                    ____w4r3z w4g0n with fr3sh 0-day k0d3z
                   /
  /-------------------\
  |     w4r3z w4g0n    |----\ <----driver of w4r3z w4g0n wearing special AWC k4m0phl4g3
  |     fr3sh 0-day    | [ ] |     kl04k1ng d3v1c3
  \____________________|_____|
    \__/     /|\         \__/ <---- tires of w4r3z w4g0n in special 0kt4g0n format
           ^ O ^
                \_____Bill Gates being dragged on the street by w4r3z w4g0n because he
                      tryed to steal 0-day k0d3z and must be punished
```

EOF
