INF-2201: 23V Operating system fundamentals

Assignment 1

Tarek Lein

UiT id: Tle044@uit.no

GitHub username: TrakeLean

January 25, 2023

1 Introduction

In this project we were assigned the task of creating a bootloader in Assembly and an image creation tool in C. with these two codes and a few precode files, we're supposed to be able to create an image that will boot up and load a kernel on a clean pc.

2 Technical Background

The biggest topic within this project was the ELF file type, the elf file has two versions it can be, it's either an executable or a linker. We were only working with the executable type.

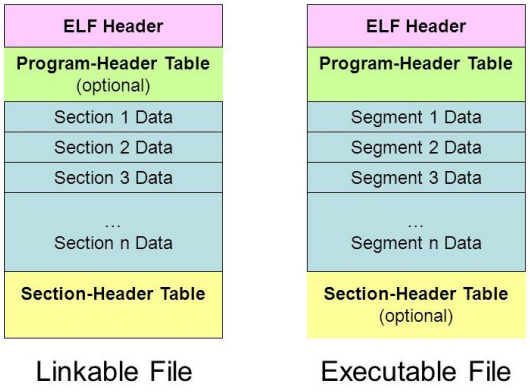

Figure 1: Elf structures

In figure 1 you can see where parts of the information is located within the elf file. In the Elf header you'll be able to find all the general information about the file it self which we'll need later, next up we find the program-header table which gives us information about address placements and the size of the file, it's important that we save the file size and use it later in the code as i will explain later in the design section. With information from these two sections we're able to read the code from the next part of the file.

The next part is where we'll find the code, which is what we are looking for, this code part consists of many different sections.

The last part is the section header table, i haven't used this in my project but I've seen ways of incorporating this part as a way to search for the code, but i think that it's an ineffective way of finding the code, so I wont go further into that method.

3 Design

I'd like to think that i implemented the easiest and most optimal way of reading the files and writing the binary data over to my own image file.

I start my program by reading the ELF header and store the variable "Section header string number index" which tells me how many sectors the file contains, i then read the program header to get access to the program header offset and the file size, these are the only things i use from these headers to read my code.

By understanding how the ELF files are structures i concluded that by the time i was done reading ELF-header and Program-header my reading pointer was at the perfect location for reading the code. Then i simply read from where i am and "size_file" forward which gives me the all the code needed, then i simply write that segment over and calculate how much the last sector need to be padded. A quick way i found for padding was to seek where it should end and write a single 0 there, this automatically pads everything above.

I have two functions, one for reading the bootblock and one for reading all the other executables, the reading and writing mentioned above are structures the same in both of these functions, the main difference is that the bootblock has to have these "magic" numbers at the end of the sector, which is from byte 510 to 512, these two bytes "55aa" tells the computer that this is a bootblock, and treats it as one.

4 Implementation

I used two ways of running my code since I'm on a Mac. The first way was to make my code with the help of a docker environment, but this made it hard to print out information for debugging, and i was only able to use QEMU to check if my code booted. QEMU would have been fine if it gave me any information about what just happened and the information behind it, but i was unable to locate these tools, which is why i decided to use method number two.

Method two was to ssh into the UiT server which let me code on a Linux computer located at Uit, while still using my Mac's screen, UI and keyboard. When doing this i got access to the debug prints in my terminal, i also got access to Bochs which gave me a huge boost when it came to debugging assembly code. With bochs i could see what every register contained and where my code halted.

Kernel security is an important thing to take into consideration, since the kernel has the highest valued admin access over everything on your computer. For this reason we want to make it as secure as possible, which is why I decided to perfectly pad the end of the file. Why? because if the computer want's to read a whole sector of 512b but we only filled in 418b, that would leave a gap of 94 bytes for anyone to hijack and inject some of their own malicious code. The way i calculated this was to multiply 512 with the amount of sectors in the file, then i subtracted the file size from that, lastly i did modulo 512 on the result, which ended up giving me the remaining bytes that needed to be filled.

5 Discussion

I know
that there is a way to access different parts of the ELF file by going into the section header table and reading the "sh_name"$then$"sh_addr"and"sh_offset"$tolocatewhereeachcodesegmentliesand$"$pick$"$themoutofthefileonebyone, Idon'treali$

6 Conclusion

This is the best it gets as far as my knowledge goes on these subjects, I'm happy with how it turned out after being overwhelmed at the start.

References

[1] https://wiki.osdev.org/Memory_Map_(x86)

[2] https://wiki.osdev.org/Stack

[3] https://wiki.osdev.org/

[4] https://stanislavs.org/helppc/int_13-2.html

[5] https://en.wikipedia.org/wiki/INT_13HINT_13h_AH=02h:_Read_Sectors_From_Drive