# Lab 8: Reading Material

## ELF-introduction Reading Material

You should read the task description as part of the reading material.

### What is ELF?

ELF stands for "Executable and Linkable Format", used by the Linux operating system. An ELF file contains binary data specifying the program code in machine language, as well as instructions to the OS on where and how to load the file into memory, where the actual code starts, etc.

### Technical Description

1. Read the ELF white paper: ELF - Executable and Linkable Format (for lab 8 you should read section 1: Object files - up-to and including symbol table)
2. Read the manual of the *readelf* utility (*man readelf*).
3. Study the picture in the following page (illustrates the ELF object file format).
4. Examine the *elf.h* header file provided with linux installations (can be found at /usr/include/elf.h), and locate the definitions of structures for the ELF header, section headers, and symbol table (In future labs you will also include this header in your own code, and need to access these structures).
5. You should be able to answer the following questions regarding any ELF executable/object (similar questions will appear in the final exam):

- Where in the file is the entry point specified, and what is its value?
- How many sections are there in this file?
- What is the size of .text section?
- Does the symbol "_end" occur in the file? If so, what is its type and bind, where is it mapped to in virtual memory?
- Does the symbol ".rodata" occur in the file? If so, what is its type and bind, where it is mapped to in virtual memory?
- Where in the file does the code of function "main" start?

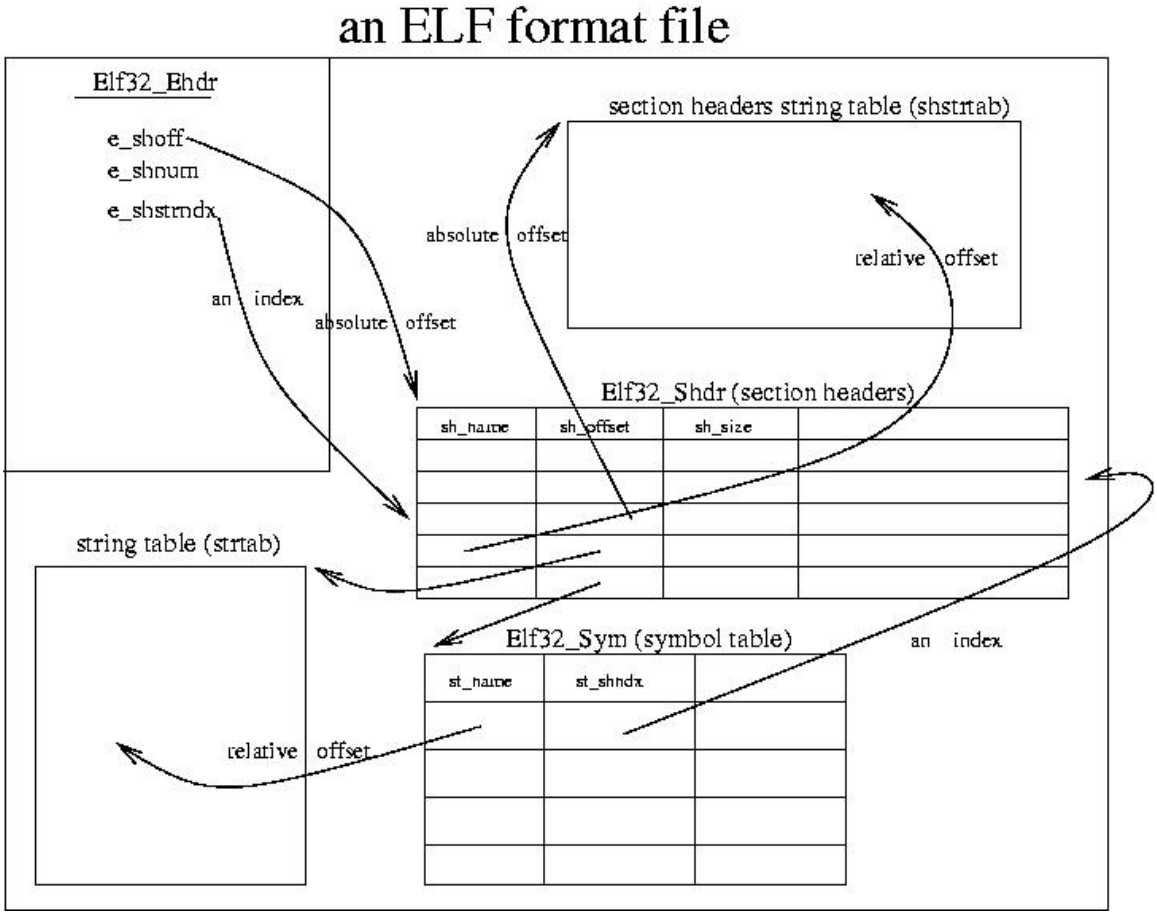
6. To get the file offset of a function, you first need to find out in which section it resides. Afterwards, the offset can be calculated from the following formula: section_file_offset + function_virtual_address - section_virtual_address. (WHY?? think about it and be prepared to explain.)

## Additional material for lab

In this lab you will be operating on files a lot. You will need to re-read on the system calls: open( ), read( ), write( ), lseek( ), and close( ). Note that lseek ( ) (see man 2 lseek) can be used to determine the size of the file using SEEK_END.

Read about the mmap system call (man mmap).

There is additional reading material about ELF in the lab zip file - also **in Hebrew**.



# Lab 8: ELF

This lab is to help you understand how your executable/relocatable files look like, and what information they contain. We will parse the ELF file and extract useful information from it. In particular, we will access the data in the section header table, and in the symbol table. We will also learn to use the `mmap` system call.

## Important

This lab is written for 32bit machines. Some of the computers in the labs already run on a 64bit OS (use *uname -a* to see if the linux OS is 64bit or not). 32bit and 64bit machines have different instruction sets and different memory layout. Make sure to include the *-m32* flag when you compile files, and to use the Elf32 data structures (and not the Elf64 ones).

In order to know if an executable file is compiled for 64bit or 32bit platform, you can use readelf, or the *file* command-line tool (for example: `file /bin/ls`).

You must only use the `mmap` system call to read/write data into your ELF files from this point onwards.

# Task 0

You should accomplish this task **BEFORE** attending the lab session.

## Task 0a

Download the following files: chezi, originalchezi.

*chezi* and *originalchezi* are both executable files in ELF. They are almost the same, except *chezi* behaves differently from *originalchezi*. Your task is to understand the reason for that.

Do the following:

1. Add execute permissions to both files: chmod u+x
2. Run the files.
3. Do they differ in size? (hint: use `ls -l *chezi`)
4. Why does their output differ? (hint: use `readelf -h`)
5. In what way is the entry point represented in the ELF file (arrangement of bytes: see ELF documentation page 9) ?

6. Now that you know where the entry point is written in the elf header and its length, open *chezi* with *hexedit* and modify it fix *chezi*, so that it behaves like *originalchezi*.

## Task 0b: Delving Deeper into the ELF Structure

Download the following file: abc. Use `readlef` with its different flags (`-h to print the header, -S to print the section table, and -s to print the symbol tables`) to do the following:

1. What is the offset of the section header table?
2. How many section headers/entries are there?
3. Find the offset of the shstrtab.
4. Find the offset (file location) of the function `main`. (hint: use `readelf -s` and `readelf -S`)
5. Find the size of the function `main`.
6. Which section does the main belong to?

## Task 0c:

This task is about learning to use the `mmap` system call. Read about the `mmap` system call (`man mmap`).

Write a program that uses the `mmap` to examine the header of a 32bit ELF file (include and use the structures in elf.h).

```
examine INFILE
```

The INFILE argument is the name of the ELF file to be examined. All file input should be read using the `mmap` system call. You are NOT ALLOWED to use `read`, or `fread`.

To make your life easier throughout the lab, map **the entire file** with one `mmap` call.

Your program should output:

1. Bytes 1,2,3 of the magic number (in ASCII)
2. Entry point (in hexadecimal)

Check using *readelf* that your data is correct.

Once you verified your output, extend *examine* to print the following information from the header:

1. Bytes 1,2,3 of the magic number (in ASCII). Henceforth, you should check that the number is consistent with an ELF file, and refuse to continue if it is not.
2. The data encoding scheme of the object file.
3. Entry point (hexadecimal address).
4. The file offset in which the section header table resides.
5. The number of section header entries.
6. The size of each section header entry.
7. The index of the shstrtab in the section header.

The above information should be printed in the above exact order (Print it as nicely as *readelf -h* does). When invoked on a non-ELF 32 bit file, your program should print an error message, and exit.

## Task 1 - Sections

Write a program which prints all the Section names in an 32bit ELF file (like `readelf -S`). Your program should exit if invoked on a non 32bit ELF file.

Your program should go over all sections in the sections table, and for each one print its index, name, address, offset, and size in bytes.

The format should be:

```
[index] section_name section_address section_offset section_size
[index] section_name section_address section_offset section_size

[index] section_name section_address section_offset section_size

....
```

Verify your output is correct by comparing it to the output of *readelf*.

You can test your code on the following file: a.out.

**Hints:** global information about the ELF file is in the ELF header, including location and size of important tables. The size and name of the sections appear in the section header table. Recall that the actual name **strings** are stored in an appropriate **section** (.shstrtab for section names), and not in the section header!

## Task 2 - Symbols

Using your program from task 1 as a starting point, you should now write a program which prints all the symbol names in a 32bit ELF file, notice that there might be more than once symbol table, and that you need to print them all. For each symbol, print

its index number, its name and the name of the section in which it is defined. (similar to `readelf -s`).

The format should be:

```
[index] value section_index section_name symbol_name

[index] value section_index section_name symbol_name

[index] value section_index section_name symbol_name

...
```

Verify your output is correct by comparing it to the output of *readelf -s*.

Your program should exit if invoked on a non 32bit ELF file.

## Task 3: Patching a program

A student was asked to write a C program that prints out the numbers in the Fibonacci sequence that are smaller than the given command line parameter N.

Having not read the instructions carefully, he instead wrote a program that prints the first N numbers in the Fibonacci sequence. To his great dismay and frustration, he also lost the source code of this wonderfully written and immensely sophisticated program.

Your mission, should you choose to accept it, is to fix the program by patching it.

Please follow the instructions below;

1. Write a program called *patch* (use the name *patch.c* for the source) which patches a file by copying a block of binary data from one file to another using ***mmap*** and ***memcpy***. The program specification is as follows:

NAME
patch – patch a program

SYNOPSIS
patch ***source_file source_pos size dest_file dest_pos***

DESCRIPTION
Copy ***size*** bytes (decimal) from position ***source_pos*** (hex) in the file ***source_file*** to position ***dest_pos*** (hex) in the file ***dest_file***.

2. Write a new C program (*task3.c*) with a proper *fib* function.

- The *fib* function in the program calculates and prints the Fibonacci sequence and returns that last value printed or -1 if nothing was printed. It has the following prototype: ***int fib(int limit);***

3. Test your program.

4. Figure out the location and size of the function *fib* in the *fibonacci* program that was written by the student.

5. Figure out the location and size of the function *fib* in your program.

6. Use the program *patch* to patch the student's program and make sure now runs correctly (what else changed in the fixed program…?)

- Notes:

1. The *fibonacci* program executable is in ***lab8_files.zip***
2. Compile your code with the flags: ***-m32 -fno-pie***
3. Make sure the code of the fib function is not too long… (what can happen if this is the case?)
4. You may choose not to accept this mission (in which case you will get 0 points for it)…

## Submission

- Submit a zip file containing your c implementation (task0.c, task1.c, task2.c, task3.c, patch.c) and makefiles to compile them.