Labs 8: ELF-Introduction with Code

This lab may be done either solo or in pairs.

In the previous lab, you learned to investigate and change ELF files using hexedit, and other command-line tools. In this lab, you will continue to manipulate ELF files, this time using your own code (written in C).

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`).

Useful Tips

You will no longer be using *hexedit* to process the file and strings to find the information; nevertheless, in some cases you may still want to use these tools for debugging purposes. In order to take advantage of these tools and make your tasks easier, you should:

- Print debugging messages: in particular the offsets of the various items, as you discover them from the headers.
- Use *hexedit* and *readelf* to compare the information you are looking for, especially if you run into unknown problems. *hexedit* is great if you know the exact location of the item you are looking for.
- Note that while the object files you will be processing will be linked using *ld*, and will, in most cases, use direct system calls in order to make the ELF file simpler, there is no reason why the programs you write need use this interface. You are allowed to use the standard library when building your own C programs.
- In order to preserve your sanity, even if the code you MANIPULATE may be without stdlib, we advise that for your OWN CODE you DO use the C standard library!
- In order to keep sane in the following lab as well, **understand** what you are doing and **keep track** of that and of your code, as you will be using them in a future lab.

Lab 8 Tasks

Deliverables
You should read and understand the reading material, and do task 0 before attending the lab. To be eligible for a full grade, you must complete tasks 1 and 2 during the regular lab. Task 3 may be done in a completion lab, if you run out of time.

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

Task 0

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). The program is first activated as:

```
myELF
```

The program then uses a menu similar to lab 7, with available operations, as follows:

```
Choose action:
0-Toggle Debug Mode
1-Examine ELF File
2-Quit
```

Note that the menu should use the same technique as in labs 2 and 7, i.e. an array of structures of available options. Toggle Debug Mode is as in Lab 7. Quit should unmap and close any mapped or open files, and "exit normally". Examine ELF Files queries the user for an ELF file name to be used and examined henceforth. 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.

In Examine ELF File, after getting the file name, you should close any currently open file (indicated by global variable Currentfd) open the file for reading, and then print the following:

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 file offset in which the program header table resides.
8. The number of program header entries.
9. The size of each program header entry.

The above information should be printed in the above exact order (Print it as nicely as *readelf* does). If invoked on an ELF file, examine should initialize a global file descriptor variable Currentfd for this file, and leave the file open. When invoked on a non-ELF file, or the file cannot be opened or mapped at all, you should print an error message, unmap the file (if already mapped) close the file (if already open), and set Currentfd to -1 to indicate no valid file. You probably also should use a global map_start variable to indicate the memory location of the mapped file.

Task 1 - Sections

Extend your myELF program from Task 0 to allow printing of all the Section names in an 32bit ELF file (like `readelf -S`). That is, modify the menu to add a "Print Section Names" option:

```
Choose action:
0-Toggle Debug Mode
1-Examine ELF File
2-Print Section Names
3-Quit
```

Print Section Names should visit all section headers in the section header table, and for each one print its index, name, address, offset, and size in bytes. Note that this is done for the file currently mapped, so if Currentfd is invalid, just print an error message and return.

The format should be:

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

[index] section_name section_address section_offset section_size  section_type

....
```

Verify your output is correct by comparing it to the output of *readelf*. In debug mode you should also print the value of the important indices and offsets, such as shstrndx and the section name offsets.

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

Extend your myELF program from task 1 to support an option that displays information on all the symbol names in a 32bit ELF file. Your menu should now be:

```
Choose action:
0-Toggle Debug Mode
1-Examine ELF File
2-Print Section Names
3-Print Symbols
4-Quit
```

The new Print Symbols option should visit all the symbols in the current ELF file (if none, print an error message and return). For each symbol, print its index number, its name and the name of the section in which it is defined. (similar to `readelf -s`). 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*. In debug mode you should first print the size of each symbol table, the number of sybmols therein, and any other useful information.

You should finish everything up to here during the lab. The rest can be done in a completion lab, if you run out of time.

Task 3 - Linker Pass I

In pass I of the linker (*ld*), it checks the symbols in all object files, and verifies that all the information it needs is available in the files.

In this task, you will implement this functionality of the linker. You should extend myELF to support this feature:

```
Choose action:
0-Toggle Debug Mode
1-Examine ELF File
2-Print Section Names
3-Print Symbols
4-Link to
5-Quit
```

The *Link to* option checks whether the current ELF file can be linked to another ELF file. The second ELF file name should be prompted from the user, opened, mapped, and checked for being a 32 bit ELF. If these steps fail, print an error message, unmap and close the second file, and return. In debug mode you should print the name of the second file, and a message prior to each step in the process (open, map, etc.).

If successful, you should then be adding appropriate tests as indicated in the following subtasks.

Simplifying Assumptions:

You may use an array for symbols, and may assume that the number of symbols is less than 10000. Handling any number of files and an unbounded number of symbols will be highly appreciated.

Task 3a: _start Function

At least one of the linked files should contain a symbol for the `_start` function. If such a symbol exists, your program should write:

```
_start check: PASSED
```

Otherwise, it should write:

```
_start check: FAILED
```

To test your program, you can use the following object files: has_start.o , no_start.o . The first should pass the test, and the second should not.

Task 3b: Duplicate Symbols

Each symbol should be defined only once in all the files. Add this check to your program. For each symbol defined in more than one object file, your program should print:

```
duplicate symbol: (symbolname)
```

otherwise (no duplicate symbols), your program should print:

```
duplicate check: PASSED
```

You can test your program with the following files:

- b1.o
- b2.o
- b3.o

Applying "Link to" to b1.o and b3.o should fail (why?), while applying "Link to" to b1.o and b2.o should pass (why?)

Task 3c: Undefined Symbols

If a symbol is needed (in the symbol table, but is "undefined") in one of the object files, it should be present in one of the others. Your "Link to" option should verify that.

For each UNDEFINED symbol appearing in one of the object files, you should verify that it is defined in one of the object files. Otherwise, this symbol is considered missing (undefined). For each missing symbol, your program should print:

```
undefined symbol: (symbolname)
```

otherwise (no missing symbols), your program should print:

```
no undefined symbols
```

You can test your program with the following files:

- c1.o
- c2.o

Link to should fail for c1.o with c2.o (why?), but should pass for b1.o with b2.o (why?)

Deliverables:

Tasks 1, and 2 must be completed during the regular lab. Task 3 may be done in a completion lab, but only if you run out of time during the regular lab. The deliverables must be submitted until the end of the day.
You must submit source files for tasks 1, 2 and task 3 and a makefile that compiles them. The source files must be named task1.c task2.c, task3.c, and makefile.