Labs 8: ELF-introduction with code

This lab may be done either alone or in pairs.

In the previous lab, you learned to investigate and change the ELF format using hexedit and other commandline tools. In this lab, we'll do similar things but using C code.

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 use the Elf32 datastructures (and not the Elf64 ones).

In order to know if an executable file is compiled for 64bit or 32bit platform, you can use the `file` commandline 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 future lab.

Lab 8 tasks

IMPORTANT: you must use only the "mmap" system call to read/write data into your ELF files from this point onwards.

Task 0

Download elf.h.

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` system to examine the header of a 32bit ELF file (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`.

Your program should output:

1. bytes 2,3,4 of the magic number (in ASCII)
2. Entry point (in hexadecimal)

Check using "readelf" that your data is correct.

Then, write a C program which gets as a command-line argument the name of an ELF-format file and prints the following information from the header (all addresses should be in hexadecimal, unless stated otherwise):

1. 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. Type
3. Entry point
4. e_shstrndx
5. The number of section header entries
6. The number of program header entries
7. The file offset in which the section header table resides.
8. Is this a 64bit or 32bit ELF file.

When invoked on a non-ELF file, your program should print an error message, and exit.

Your program code should include elf.h, which contains all the typedefs and constant definitions for handling ELF files.

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 name, index, type and size in bytes.

The format should be:

```
[index] section_name section_type section_size
[index] section_name section_type section_size

[index] section_name section_type section_size

....
```

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

You can test your code on the following file: cat.

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. 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] symbol_name section_name

[index] symbol_name section_name

[index] symbol_name section_name

[index] symbol_name section_name

...
```

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

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

Hints: Symbols are listed in the .symtab section. The section in which a symbol is defined (if it is defined) is the index of the symbol, which is an index into the section header table, referring to the section header of the appropriate section, and from there the section name can be retrieved as above. Symbol name is an attribute of the symbol structure, but recall again that the actual name string is stored in a string table, a separate section (.strtab).

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 - Can the files be linked?

Before the linker (`ld`) tries to link object files, it 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 write a program, `can_link` getting as arguments 2 32bit object files.

```
can_link file1 file2
```

Each of the following tasks will add a check to your program.

If all checks pass, your program should exit with 0 return code.
If a check failed, you should print the reason for the failure and exit with a -1 return code.

If one of the files is not a 32bit object file, your program should print an error message and exit with an exit code of -2.

Simplifying assumptions: you may use an array for symbols, and may assume that the number of symbols is less than 10000. Small bonus for handling any number of files and unbounded number of symbols.

Task 3a: main function

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

```
main check: PASSED
```

Otherwise, it should write:

```
main check: FAILED
```

and exit with a code of -1.

To test your program, you can use the following object files: hasmain.o, nomain.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. If the same symbol is defined in more than one object file, your program should print:

```
duplicate check: FAILED (symbolname)
```

and exit with a code of -1.

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

```
duplicate check: PASSED
```

(think: which piece of code from previous labs can help you in this task?)

You can test your program with the following files:

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

```
can_link b1.o b3.o
```

should fail (why?), while

```
can_link b1.o b2.o
```

should pass (why?)

Task 3c: undefined symbols

If a symbol is needed (in the symbol table, but "undefined") in one of the object files, it should be present in one of the others. Your program 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). If you find a missing symbol, your program should print:

```
no missing symbols: FAILED (symbolname)
```

and exit with a code of -1.

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

```
no missing symbols: PASSED
```

You can test your program with the following files:

- c1.o
- c2.o

```
can_link c1.o c2.o
```

should fail (why?), while

```
can_link b1.o b2.o
```

should pass (why?)