Lab 7: ELF-introduction

(This lab is to be done SOLO and NO completion labs will be held)

Lab Goals

- Extracting useful information from ELF files (under 64-bit architecture).
- Fixing files using this information - reverse engineering.

In the following labs, you will learn to handle object and executable files. We will begin by learning just some of the basics of ELF, together with applications you can already use at this level - editing binary files and writing software patches. Then, we will continue our study of ELF files, by beginning to parse the structures of ELF files, and to use them for various purposes. In the next lab, we will access the data in the section header table and in the symbol table.

Methodology

- Get to know the ELF.
- Learn how to use the *readelf* utility. By using *readelf* you can get, in a human readable format, all the ELF structural information.
- Experience basic ELF manipulation.

Recommended Operating Procedure

This advice is relevant for all tasks. Note that while at some point you will no longer be using *hexedit* to process the file and *readelf* to get 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:

- You may use debugging messages: in particular the offsets of the various items, as you discover them from the headers. Also, whenever the user is required to enter values, you should print the **parsed** values in their respective representation (e.g. string, decimal or hexadecimal).
- 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 the system_call 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! (Yes, this is repeated twice, so that you notice it!)

All the executable files we will work with in this session are 64-Bit ELF binaries.
Compile your code accordingly.

Lab 7 Tasks

You should read and understand the reading material, and do task 0 before attending the lab.

Task 0

Task 0a: Memory Leaks, Segmentation Faults, and Printing data from files in hexadecimal format

- Programs inevitably contain bugs, at least when they are still being developed. Interactive debugging using `valgrind(1)` helps locate and eliminate bugs. `valgrind` assists in discovering illegal memory access even when no segmentation fault occurs (e.g. when reading the n+1 place of an array of size n). `valgrind` is extremely useful for discovering and fixing memory leaks. It will tell you which memory allocation was not freed.

  To run valgrind write: `valgrind --leak-check=full --show-reachable=yes [program-name] [program parameters]`.
  If valgrind reports errors in your code, repeat the execution with the "-v" flag like so:
  `valgrind -v --leak-check=full --show-reachable=yes [program-name] [program parameters]`.

  The source code of a buggy program, bubblesort.c, is provided. The program should sort numbers specified in the command line and print the sorted numbers, like this:

```
$ bubblesort 3 4 2 1
Original array: 3 4 2 1
Sorted array: 1 2 3 4
```

  However, an illegal memory access causes a segmentation fault (segfault). In addition, the program has a few memory leaks.
  First solve the segfault using `gdb` (or just by reading the code). Then use valgrind to find the memory leaks and fix them.

Task 0b

Download the following file: a.out. Answer the following questions (be prepared to explain your answers to the lab instructor):

1. Where is the entry point specified, and what is its value?
2. How many sections are there in a.out?
3. What is the size of the .text section?
4. Does the symbol _start occur in the file? If so, where is it mapped to in virtual memory?
5. Does the symbol main occur in the file? If so, where is it mapped to in virtual memory?
6. Where in the file does the code of function "_start" start?

Task 0c

Write a program called *hexeditplus*:

```
./hexeditplus
```

The hexeditplus program performs operations (read and write) on files and memory. File operations are done on a file *filename* as defined below. Each operation is done in units of *size* bytes, which indicates a unit size, i.e. the number of bytes we want to use as the basic unit in each operation of our program, such as "display file contents". Size can be either 1, 2 or 4, with 1 as the default.

The variables *size* and *filename* (initially null) should be global variables.

First, define a menu for the user with a number of predefined functions, to which we will add functions as we go. The program prints the menu, obtains a choice from the user, acts on it, and repeats infinitely. For example, if the functions: "Set File Name", "Set Unit Size" and "Quit" are available, then the command line:

```
./hexeditplus
```

Will print:

```
Choose action:
1-Set File Name
2-Set Unit Size
3-Quit
```

For this part, use an array with the above menu names and pointers to appropriate functions that implement each option.

At this point implement "Set File Name", "Set Unit Size", and "Quit".

Set File Name queries the user for a file name, and store it in a globally accessible buffer. You may assume that the file name is no longer than 100 characters.

The Set Unit Size option sets the size variable. The steps are:

1. Prompt the user for a number.
2. If the value is valid (1, 2, or 4), set the size variable accordingly.
3. If not valid, print an error message and leave size unchanged.

Quit is a function that calls `exit(0)` to quit the program.

The rest of the functions will be written in the next tasks. The menu should be extensible, you will change and extend it in each sub-task of task 1. It should be printed using a loop iterating over the menu array, and be {NULL, NULL} terminated.

All functions should be of the form:

```
void func( );
```

Be sure to implement this code and test it carefully before the lab (that is why you have the debug option), as you will need to extend it during the lab!

Task 1: hexeditplus

In this task we will write our own version of *hexedit* for working with binary files. You will extend your code from task 0c.
Note: You should verify that there is no error when opening a file. In case of an error, you should print a message and abort the rest of the operation.
For this task you will be working with the following ELF file: test.

Task 1a: File Display

Write the function for the "File Display" option:
This option displays *length* units from the file *filename* (chosen using option 1 in the menu), starting at file location *location* (note: this is the same as the "offset" in the file). The units should be displayed once using a hexadecimal representation, and again using a decimal representation.
The steps are:

1. Check if *filename* is null, and if it is print an error message and return.
2. Open *filename* for reading. If this fails, print an error message and return.
3. Prompt the user for location (in hexadecimal) and a length (in decimal).
4. Allocate *<unit size>* * *<length>* bytes on the heap.
5. Read *length* units from file *filename*, starting from *location*, into the allocated memory.
6. Close the file.
7. Display in **hexadecimal**, the read units.
8. Display in **decimal**, the read units
9. Free the allocated memory
   - Note that you should only read from the file once in order to implement both prints (decimal and hexadecimal).

For example, the command line:

```
./hexeditplus
```

Will print:

```
Choose action:
1-Set File Name
2-Set Unit Size
3-File Display
4-Quit
```

After the user set the unit size to 2 and chooses 3 with *location* "40" and *length* "5" - your program will print the 5 units, starting from the location 0x40, and ending in location 0x49.

The prompt should look as follows:

```
3
```

```
Please enter <location> <length>
110 6
```

If the current file is "a.out", then the output should look something like this (for a unit size of 2):

```
Hexadecimal Representation:
485f e689 8948 48ca e2c1 4802
Decimal Representation:
18527 59017 35144 18634 58049 18434
```

- Note that, depending on the chosen unit size, the printed hexadecimal values may differ in order when compared with the output of *hexedit*. Why is that?

Use your newly implemented functionality to answer: what is the entry point of your own *hexeditplus* program? Verify your answer using `readelf -h`

Remember

- To read *location* and *length* use `fgets` and then `sscanf`, rather than `scanf` directly.
- *location* is entered in hexadecimal representation.

Task 1b: File Modify

Write the function for the "File Modify" option:
This option replaces a unit at location in file filename with val.
The steps are:

1. Prompt the user for *location* and *val* (all in hexadecimal).
2. Replace a unit at *location* in the file with the values given by *val*.

For example, the command:

```
./hexeditplus
```

Will print:

```
Choose action:
1-Set File Name
2-Set Unit Size
3-File Display
4-File Modify
5-Quit
```

When the user chooses option 4, the program should query the user for:

- *location* (file location, in hexadecimal)
- *val* (new value, in hexadecimal)

For example, if unit size was set to 4, choosing option "4-File Modify" using *location* 0x40, *val* 0x804808a, will overwrite the 4 bytes starting at location 0x40, with the new value 804808a. It should look as follows:

```
4
Please enter <location> <val>
40 804808a
```

Resulting in the file changing like so:

```
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
8a 80 04 08 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
```

You should check that the file is opened correctly and that the location chosen to be modified, given the current unit size, is valid, and act accordingly.

You can test the correctness of your code using task1a - "File Display" (the displayed result should be the same as the value given, if the units are the same).

Task 1c: Copy From File

Write the function for the "Copy From File"

- Check if *src_file* or *dst_file* is null, and if it is print an error message and return.
- Open *src_file* for reading. If this fails, print an error message and return.
- Open *dst_file* for read & write (**not** write only). If this fails, print an error message and return.
- Prompt the user for *src_offset* and *dst_offset* (in hexadecimal) and *length* (in decimal).
- Read *length* bytes from *src_file* at offset *src_offset* and write them to *dst_file* at *dst_offset*
- Close the files.

For example, the command:

```
./hexeditplus
```

Will print:

```
Choose action:
1-Set File Name
2-Set Unit Size
3-File Display
4-File Modify
5-Copy From File
6-Quit
```

Assume that the user has already set the *dst_file* name to "test". If the user chooses 5, he is prompted for *src_file*, *src_offset*, *dst_offset*, and *length*. It should look as follows:

```
5
Please enter <src_file> <src_offset> <dst_offset> <length>
abc2 12F 12E 10
```

The program should open the file "abc2" and load the 10 bytes, from byte 303 to byte 312 into file "test". The output should look like:

```
Copied 10 bytes into FROM abc2 at 12F TO abc at 12E
```

Task 2: Reading ELF

Task 2a

Download the following file: buggy.

`buggy` is an executable ELF file. It does not run as expected. Your task is to understand the reason for that.

Do the following:

1. Run the file.
2. Which function precedes main in execution ? (hint: The assembly code in Lab 4).
3. What is the virtual address to which this function is loaded (hint: use `readelf -s`)

Task 2b

Use your *hexeditplus* program from task 1 to display the entry point of a file.

What are the values of *location*/*length*? How do you know that?

Use the edit functions from *hexeditplus* program to fix the *buggy* file, so that it behaves as expected.

Task 3: Hacking: Installing A Patch Using hexeditplus

Task 3a: Delving Deeper Into The ELF Structure

The goal of this task is to display the compiled code (in bytes) of the function `main`, in the *test* executable above.

In order to do that, you need to:

1. find the offset (file location) of the function `main`.
2. find the size of the function `main`.
3. use your *hexeditplus* program to display the content of that function on the screen.

Finding the needed information:

1. Find the entry for the function `main` in the symbol table of the ELF executable (`readelf -s`).
2. In that reference you will find both the size of the function and the function's virtual address and section number.
3. In the section table of the executable, find the entry for the function's section (`readelf -S`).
4. Find both the section's virtual address (Addr), and the section's file offset (Off).
5. Use the above information to find the file offset of the function.

Task 3b

The following file ntsc was meant to be a digit counter. Download it, and run it in the command-line.

```
./ntsc aabbaba123baacca
./ntsc 1112111
```

What is the problem with the file? (hint, try this string: 0123456789)

Create a new program with a correct digit counter function (should get a char* and return an int), compile and test it. (remember to compile with the -m64 flag in order to produce an ELF compatible with 64bits).

Use *hexeditplus* to replace (patch) the buggy `digit_cnt` function in the *ntsc* file with the corrected version from the new program.
You should do it using option 5 in *hexeditplus*.
(think: are there any kinds of restrictions on the code you wrote for the `digit_cnt` function?)
Explain how you did it, and show that it works.

Deliverables:

There are no completion tasks in this lab. All tasks must be completed during the lab session

The deliverables must be submitted until the end of the day.
You must submit source files for task 1 and a makefile that compiles it. The source files must be named task1.c and makefile.

Submission instructions

- Create a zip file with the relevant files (only).
- Upload zip file to the submission system.
- Download the zip file from the submission system and extract its content to an empty folder.
- Compile and test the code to make sure that it still works.