

Lab 4: ELF - introduction

This lab **may** be done in pairs.

You must complete task 0 before attending the lab, and we also recommend doing tasks 1a, 1b, and 1c before the lab.

Lab Goals

- Extracting useful information from files in ELF.
- Fixing files using this information: reverse engineering.

In the following labs, you will learn to handle object and executable files. We 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 particular, we will access the data in the section header table and in the symbol table.

Methodology

- 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:

- Support 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 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! (Yes, this is repeated twice, so that you notice it!)
- In order to keep sane in the following labs as well, **understand** what you are doing and **keep track** of that and of your code, as you will be using them in future labs.

All the executable files we will work with in this session are 32-Bit ELF binaries. Compile your code accordingly.

Lab 4 Tasks

Task 0

Task 0a:

Download the executable file *abc* (from the [Lab 4 auxilliary files](#)), and 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 "abc"?

Task 0b

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 *file_name* 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 memory contents". Size can be either 1, 2 or 4, with 1 as the default.

First, define a menu for the user with a number of predefined functions (as done in Lab 1), 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. Specifically, the command line:

```
./hexeditplus
```

Should print:

Choose action:

0-Toggle Debug Mode

1-Set File Name

2-Set Unit Size

3-Load Into Memory

4-Toggle Display Mode

5-Memory Display

6-Save Into File

7-Memory Modify

8-Quit

For this part, use an array with the above menu names and pointers to appropriate functions that implement each option, using the same menu scheme from lab 1.

At this point implement "Toggle Debug Mode", "Set File Name", "Set Unit Size", and "Quit". All other functions should at this point be stubs that print "Not implemented yet" and return without doing anything else.

Toggle debug mode means turn the debug flag on (if it is currently off, which it is in the initial state), and print "Debug flag now on". If the debug flag is on, this function prints "Debug flag now off", and turns the flag off. When the debug mode is on, you should print the value of the variables: `unit_size`, `file_name`, and `mem_count`, every time just before the menu is printed.

All functions should be of the form:

```
void fun(state* s); // Getting the state as a pointer allows the functions
to change it.
```

where the state struct is defined as:

```
typedef struct {
    char debug_mode;
    char file_name[128];
    int unit_size;
    unsigned char mem_buf[10000];
    size_t mem_count;
    /*
     *
     * Any additional fields you deem necessary
     */
} state;
```

Set File Name queries the user for a file name, and store it in *file_name*. You may assume that the file name is no longer than 100 characters. If debug mode is on, the function should also print (to stderr, as are all debug messages): "Debug: file name set to 'file_name' " (obviously, replacing 'file_name' with the actual name).

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 debug mode is on, print "Debug: set size to x", with x the appropriate size.
4. If not valid, print an error message and leave size unchanged.

Quit is a function that prints "quitting" (in debug mode), and 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.

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!

Implementation note: working with units

Later on, in task 1, you are required to write code that handles data in unit sizes (i.e. not necessarily single bytes). This might confuse you into writing much more code than needed. See below how to handle multiple unit sizes when reading, printing etc. without writing too much code. Make sure you look at this code and understand how to use it before attending the lab. As a minimum, temporarily make the display-memory stub print a constant number (e.g. 0x5555) using the current size, in decimal and hexadecimal, as an exercise. Relevant to this task is the function `print_units`, and also see below.

Note that, you can use the following lines to use for printing

```
printf(dec_formats[u-1], val);  
printf(hex_formats[u-1], val);
```

where *u* is the current unit size and *val* is the val that we want to print, and with the arrays defined as follows:

```

        static char* hex_formats[] = {"%#hhx\n", "%#hx\n", "No such
unit", "%#x\n"};
        static char* dec_formats[] = {"%#hhd\n", "%#hd\n", "No such
unit", "%#d\n"};

```

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 0b.

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: *abc* (see [Lab 4 auxiliary files](#)).

Note: For any functions that handle files, the file needs to be opened and closed **within that function**

Task 1a: Load Into Memory

For this task, you may assume that the size of the data being loaded is less than 10000 bytes.

Write the function for the "Load Into Memory" option, which works as follows:

- Check if *file_name* is empty (i.e. equals to ""), and if it is print an error message and return.
- Open *file_name* for reading. If this fails, print an error message and return.
- Prompt the user for *location* in **hexadecimal**, and *length* (in decimal).
- If debug flag is on, print the *file_name*, as well as *location*, and *length*.
- Copy *length * unit_size* **bytes** from *file_name* starting at position *location* into *mem_buf*.
- Close the file.

Assume that the user has already set the file name to "abc". If the user chooses 3 on the menu, he is prompted for *location* and *length*. It should look as follows:

```

3
Please enter <location> <length>
12F 10

```

The program should open the file *abc* and load the 10 bytes (assuming unit size is set to 1), from byte 303 (which is the decimal value of 0x12F) to byte 312 in the file into *mem_buf*. The output should look like:

Loaded 10 units into memory

Remember

- To read *location* (hexadecimal) and *length* (decimal) use `fgets` and then `sscanf`, rather than `scanf` directly.
- Note again that *location* is always entered in hexadecimal representation.

Task 1b: Toggle Display Mode

Write the function for the "Toggle Display Mode" option, which switches between display using a decimal representation, and display using a hexadecimal representation.

Toggle display mode means turn the display flag on and print using a hexadecimal representation, the initial state is off and print using a decimal representation. Print "Display flag now on, hexadecimal representation", and if the display flag is on, this function prints "Display flag now off, decimal representation", and turns the flag off. For example, (assume that the display flag is off) entering option 4 in the menu:

```
4
Display flag now on, hexadecimal representation
4
Display flag now off, decimal representation
```

Task 1c: Memory Display

Write the function for the "Memory Display" option:

This option displays *u* units of size *unit_size* starting at address *addr* in memory. *Unit_size* is already defined in *state*, but *u* and *addr* should be queried from the user by this function. *u* will be given in decimal and *addr* in hexadecimal. Entering a value of 0 for *addr* is a special case, in which the memory to be displayed starts at your `mem_buf`.

The units should be displayed according to the display flag. If the display flag is on then print using a hexadecimal representation, and if the display flag is off print using a decimal representation.

If the user set the unit size to 2 and loaded a file into memory, then the output should look something like this (remember, we previously loaded 5 pairs of bytes from the file *abc*, located 303-312):

```
Choose action:
0-Toggle Debug Mode
1-Set File Name
2-Set Unit Size
3-Load Into Memory
4-Toggle Display Mode
5-Memory Display
6-Save Into File
7-Memory Modify
8-Quit
> 5
Enter address and length
> 0 5
Decimal
=====
256
0
12032
26988
12130
```

```
Choose action:
0-Toggle Debug Mode
1-Set File Name
2-Set Unit Size
3-Load Into Memory
4-Toggle Display Mode
5-Memory Display
6-Save Into File
7-Memory Modify
8-Quit
> 4
Display flag now on, hexadecimal representation
Choose action:
0-Toggle Debug Mode
1-Set File Name
2-Set Unit Size
3-Load Into Memory
4-Toggle Display Mode
5-Memory Display
6-Save Into File
7-Memory Modify
8-Quit
> 5

Enter address and length
> 0 5
Hexadecimal
=====
100
```

0
2F00
696C
2F62

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 (load into memory and memory display) to answer: what is the entry point of your own *hexeditplus* program? Verify your answer using `readelf -h`

Task 1d: Save Into File

Write the function for the "Save Into File" option, which works as follows:

This option replaces *length* units (each of size determined by the current unit size) at *target-location* of *file_name* with bytes from the **hexeditplus** memory starting at virtual address *source-address*. Note that the filename is the last file name set by option 1 "Set File Name".

When the user chooses option 6, the program should query the user for:

- *source-address* (source **virtual memory** address, in hexadecimal), *source-address* can be set to 0, in which case, the source address is start of *mem_buf*, in any other case, use *source-address* as an address in (virtual) memory.
- *target-location* (target file offset, in hexadecimal),
- *length* (number of units, in decimal).

Implement the checks that the file can be opened (for writing and NOT truncating), and print appropriate debug messages in debug mode as in the previous task. Close the file after writing.

For example, after the file name was set to "abc" and unit size to 1 bytes, choosing option "6-Save Into File" using *source-address* 960c170, *target-location* 33 and *length* 4, the program should write *length* = 4 bytes from (virtual) memory,

starting at address 0x960c170 to the file *abc*, starting from offset 0x33 (overwriting what was originally there). It should look as follows:

```
6
Please enter <source-address> <target-location> <length>
960c170 33 4
```

Note again that the target file is the one specified using option 1 in the menu.

Also observe that after you execute this option, **only** *length* units of the file *file_name* should be changed.

If <target-location> is greater than the size of <file_name> you should print an error message and not copy anything.

You should use *hexedit*, to verify that your code for tasks 1c and 1d works correctly, by loading a portion of a file into memory and saving it to another file.

Here is some of *hexedit*'s output for the file *abc*, verify that you understand why the output is as it is.

```
00000070 01 00 00 00 01 00 00 00 00 00 00 00 00 80 04 08
.....
00000080 00 80 04 08 EC 05 00 00 EC 05 00 00 05 00 00 00
.....
00000090 00 10 00 00 01 00 00 00 14 0F 00 00 14 9F 04 08
.....
000000A0 14 9F 04 08 0C 01 00 00 14 01 00 00 06 00 00 00
.....
000000B0 00 10 00 00 02 00 00 00 28 0F 00 00 28 9F 04 08
.....(...(
000000C0 28 9F 04 08 C8 00 00 00 C8 00 00 00 06 00 00 00
(.....
000000D0 04 00 00 00 04 00 00 00 48 01 00 00 48 81 04 08
.....H...H...
000000E0 48 81 04 08 44 00 00 00 44 00 00 00 04 00 00 00
H...D...D.....
000000F0 04 00 00 00 51 E5 74 64 00 00 00 00 00 00 00 00
....Q.td.....
00000100 00 00 00 00 00 00 00 00 00 00 00 00 06 00 00 00
.....
00000110 04 00 00 00 52 E5 74 64 14 0F 00 00 14 9F 04 08
....R.td.....
00000120 14 9F 04 08 EC 00 00 00 EC 00 00 00 04 00 00 00
.....
00000130 01 00 00 00 2F 6C 69 62 2F 6C 64 2D 6C 69 6E 75
....../lib/ld-linu
00000140 78 2E 73 6F 2E 32 00 00 04 00 00 00 10 00 00 00
x.so.2.....
00000150 01 00 00 00 47 4E 55 00 00 00 00 00 02 00 00 00
....GNU.....
00000160 06 00 00 00 0F 00 00 00 04 00 00 00 14 00 00 00
.....
00000170 03 00 00 00 47 4E 55 00 C1 4E 4D 18 B9 A6 21 8F
....GNU..NM....!
```

Task 1e: Memory Modify

Write the function for the "Memory Modify" option:

This option replaces a unit at *location* **in the memory buffer** (not virtual memory address!) with *val*.

The steps are:

1. Prompt the user for *location* and *val* (all in hexadecimal).
2. If debug mode is on, print the location and val given by the user.
3. Replace a unit at *location* in the memory with the value given by *val*.

When the user chooses option 7, the program should query the user for:

- *location* (memory buffer location, in hexadecimal)
- *val* (new value, in hexadecimal)

For example, if unit size was set to 4, choosing option "7-Memory 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:

7

Please enter <location> <val>

40 804808a

As in the previous task, you should check 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 hexedit (by saving the memory buffer to a file and then looking at the resulting change using hexedit).

Task 2: Reading ELF

Download the following file: deep_thought from Lab 4 Files.

deep_thought 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" should precede main() in execution ? (Hint: see assembly code in Lab 3).

3. What is the virtual address to which this function is loaded (Hint: use `readelf -s`)

Fixing the buggy executable file

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 *deep_thought* file, so that it behaves as expected.

Task 3: 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 *offensive* executable in the lab 4 auxiliary files.

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.

Hacking the executable file

Hack this executable file so that it does nothing when it is run: replace the code of the `main` function by [NOP](#) instructions.

Make sure you do NOT override the `ret` instruction ([Opcode](#): c3) in `main`. Alternately, you can plant just one `ret` instruction (where?).

Task 4: Hacking: installing a patch using hexeditplus

The following file `ntsc` file was meant to be a digit counter. Download it from Lab 4 Files, 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 `-m32` flag in order to produce an ELF compatible with 32 bits, and also **must use the compile flags: `'-fno-pie'` and `'-fno-stack-protector'`**

Use *hexeditplus* to replace (patch) the buggy `count_digits` function in the `ntsc` file with the corrected version from the new program.

You should do it using options 3 & 6 in *hexeditplus*.

(think: are there any kinds of restrictions on the code you wrote for the `count_digits` function?)

Explain how you did it, and show that it works.

Deliverables:

All tasks must be completed during the regular lab. The deliverables must be submitted until the end of the lab session.

You must submit source files for task 1 and task 4 and a makefile that compiles them. The source files must be named `task1.c`, `task4.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.