CSC209H Worksheet: Inspecting Executables

Programs are simply binary files whose contents can be interpreted as a set of machine instructions. Since they are necessarily in a known (specified) format, we can write programs that inspect and modify them.
For instance the Unix program `objdump`[1] displays sections of executable files in human readable form. Given a compiled "hello world" program and specifying the "read-only data section", `objdump` produces:

```
$ objdump -s -j .rodata hello

hello:     file format elf64-x86-64

Contents of section .rodata:
 0700 01000200 48656c6c 6f20776f 726c6421  ....Hello world!
 0710 00
```

The read-only section of the executable is where string literals (among other things) are stored, and `objdump` lets us see them. The output above is formatted as follows:

- The first column contains memory addresses. In the example, the `.rodata` section starts at address `0700`.
- The next 4 columns contain the hexadecimal representations of the memory contents at those addresses. Recall that two hexadecimal digits together represent one byte, so one line represents 16 bytes. There are 17 bytes in the `.rodata` section of our example.
- The last column also displays the memory contents, but in ASCII representation. Some of the bytes are printable as valid ASCII characters, whereas other bytes are not (and are thus represented by a `.` instead), since the `.rodata` section can contain data besides string literals.

`objdump` behaves slightly differently on different platforms – and with different executable formats. For example, on OS X, files don't have a `.rodata` section, so you would need to omit the "-j" flag to look at all of the sections. You might also run into a situation where the offset reported is not the same as the offset in the file. In that case, you can add the "–file-offsets" flag. If you're not running on the lab machines and get odd behaviour from `objdump`, flag down a TA or instructor for help.
We're going to use the information from `objdump` to write a program that extracts data from a binary file. This will be particularly useful for assignment 2. As in that assignment, we're going to use `fseek` and binary IO, and we'll be reading header data into `structs`. We've provided some starter code, including the example program above and a `Makefile`, on the course website.

Exercise 1

Write a program `literals.c` that prints all of the string literals stored in the `.rodata` section, one per line. The program should take three command-line arguments:

1. *address in hexadecimal* of the first byte in the `.rodata` section,
2. *size (number of bytes) in decimal* of the `.rodata` section, and
3. *name* of an executable file

Notice in the sample output below that there are two lines before "Hello world!", since there is some non-string data that is stored before the literal in the `.rodata` section, as can be seen in the `objdump` output above):

```
$ ./literals 0x700 17 hello


Hello world!
```

Helpful tips:

- Use base-16 `strtol` to parse hexadecimal numbers that lead with `0x`.
- Use `fseek` to jump to the part of the exectutable file that you want to read.

[1] Objdump comes included in the GNU Binary Utilities. You may be interested in some of the other included tools, as well: `https://www.gnu.org/software/binutils/`

- Use `fread` to read in the entire `.rodata` section first. Then think about how to print out individual strings on new lines. Remember that there could be many null-terminated strings in the section. (And as above, it's okay if you print some garbage because there is non-string data stored.)

Use `hello.c` to test – you should get exactly the same output as in our example. Then, grab your A1 code or lab code and see if you can extract the literals from that source file. Finally, try your code out on `copied_hello.c`. What do you think is happening? Do you get a different result if you replace the `strcpy` with `strncpy`?

Exercise 2

For `literals` to be useful we need a program to determine the location and size of the `.rodata` section of a given executable. It should not be surprising that this data is encoded in the executable itself. Here is the relevant information **for basic programs compiled using gcc on the lab machines**:

- The four bytes at addresses `0x28` to `0x2b` contain the starting address for all the section headers.
- The file `rodata.h` contains a struct that matches the format of a ELF-64 section header. If you read data from the file into a struct, then the data will fill in the fields correctly. The `sh_addr` and `sh_size` members will contain the address and size of the header being described.
- The `.rodata` section is the 17th header. You can either seek directly to it and then read just that header, or you can read at least 17 headers (into an array of header structs) and then get the data you need from the 17th header in the array.

Create a program `rodata.c` which takes the filename of an executable and prints the address of the `.rodata` section (in hex) and the size of the section (in decimal).

```
$ ./rodata hello
0x00000700 17
```

Exercise 3

Modify `literal` to accept the output from `rodata` through a pipe, rather than requiring those values as command line arguments.

```
$ ./rodata hello | ./literals hello


Hello world!
```