CS220 - Computer System II
Lab 2

1 Introduction

In this lab, you will understand different types of memory and variable scope.

2 Getting Started

In the repository, you should find a program named `memory`. Although you will not compile `memory`, for all of the programs that you write and compile, use the flags `-g -std=c89 -O0`. The flag `-g` retains debug information in your executable, `-std=c89` tells the compiler to use the ANSI C standard, and `-O0` turns off optimization[1].

3 Different types of memory

The precompiled program `memory` has secrets. Your job is to:

- Identify the secret values (see program listing below).
- Identify the addresses of `gfoo`, `lfoo`, `sfoo` and `dfoo`. Identifying the value of `dfoo` is the address where one of the secrets will be stored.
- Identify the section where each variable is stored.

The source code of the program is as follows:

[1] If we don't do this, the compiler may realize that allocated memory is not being used, and may therefore skip the allocation altogether!

```c
#include <stdlib.h>
int gfoo = /* secret 1 */;
int main()
{
  static int sfoo;
  int lfoo;
  int *dfoo = (int *) malloc (sizeof(int));
  sfoo = /* secret 2 */
  lfoo = /* secret 3 */
  *dfoo = /*secret 4 */
  return sfoo + lfoo + gfoo + *dfoo;
}
```

Primarily, three main memory stores are available to each program. The program itself (i.e., .data, .rodata, .bss etc.), the program stack, and the program heap. Any variable that is local to a function during execution is initialized each time the control enters the function. It is stored on the stack. Any variable that persists across function invocations (e.g., global and static variables) can not be stored on the stack region. Memory allocated through `malloc` are acquired at runtime and are therefore stored on the heap.

Run the program on `gdb` and set appropriate break points. When you run the program, the program is loaded. At that point, you can examine the memory mappings and layouts using the following commands:

```
(gdb) info proc mappings
(gdb) info files
```

In order to find the secrets, you could set a breakpoint after the move instructions that occur between the call to main and return, and look at the immediate values.

Deliverable Create a `lab2.txt` file, record the following:

1. Secrets 1, 2, 3 and 4.
2. Addresses of gfoo, sfoo, lfoo and dfoo.
3. Value of dfoo (which is the address that contains secret 4).

4. Start and end addresses of stack, heap and data section.

4 Effect of type of memory on a program

4.1 Effect of global variables

In the file `global_vars.c`, modify the code and compile, each time replacing x with values 10, 100, 1000 and 10000 respectively.

```
#include <stdio.h>
char gfoo[x] = {0x10};
int main()
{
  printf("%p\n", &gfoo);

  return 0;
}
```

Deliverable For each version of `global_vars.c`, record the following in `lab2.txt`:

1. The value of x and the size of the binary.

Because global variables are incorporated into a writable section in the binary, you will find that the size of the binary increases as the number of global variables increase.

4.2 Effect of dynamic variables

In the file `malloc_vars.c`, modify the code and compile, each time replacing x with values 10, 100, 1000 and 10000 respectively.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *p = (int *) malloc (x * 0x1000)
  return 0;
}
```

Deliverable For each version of `malloc_vars.c`, record the following in `lab2.txt`:

- The value of x and the size of the binary.

Because memory is acquired at runtime, you will not find a significant change in the binary size.

4.3 Testing the limits of dynamic memory

Write your own program in a source file called `malloc_limits.c` to allocate increasing amounts of memory using malloc.

Here are some hints to consider:

- You may find it helpful to base it on `malloc_vars.c`.
- According to the *malloc(3)* manual page,

> The malloc() functions **return a pointer** to the allocated memory, which is suitably aligned for any built-in type. **On error, these functions return NULL**.

Also, insert a break point in gdb on entering main and just before returning from main. To record the size of heap at each breakpoint, use

```
(gdb) info proc mappings
```

Deliverable At some point, the program will run out of memory. Record the following in `lab2.txt`:

- Maximum amount (in bytes) of dynamic memory you can acquire.

- Size of heap at each breakpoint.

Unlike Java, C does not come with a garbage collector. Therefore, it is very important to free the dynamic memory that is not being used using the `free()` function.

4.4 Testing limits of stack memory

In the file `static_vars.c`, modify the code and compile, each time replacing x with values 10, 100, 1000 and 10000 respectively.

```c
#include <stdio.h>

int main()
{
  int arr[x];
  return sizeof(x);
}
```

Deliverable Insert a break point in gdb on entering main and just before returning from main. Use"info proc mappings" and record the following in `lab2.txt`:

- Size of heap at each breakpoint.

5 Submitting the result

1. Commit your `malloc_limits.c` and `lab2.txt` files to the repository.
2. Push any commits you've made to the remote repository.
3. Record the commit's SHA hash value in MyCourses and submit it in this lab's assignment.