ENPM691 Homework Assignment 4

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to determine the locations in memory the compiler uses for different variable declaration types using GDB automation. The script will give the commands instead of the user.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

There are four storage classes in C: auto, static, extern, and register. With these keywords, memory storage and initialization options differ. Along with that, in C where a variable is declared, meaning the scope makes a lot of difference to the compiler in terms of parsing and generating the assembly [1].

In this experiment, we will be using GNU Debugger to process the executable. Pointers will be used to check where a particular variable will be stored in memory. The goal of the assignment is to explore GDB automation through scripts for looking at the variable addresses and the way they are stored in memory. Also, why certain memory locations are allotted to certain types of declaration.

II. Methodology

A. Host environment and architecture

[1] The experiment was done on a Ubuntu v16.04 VM with Windows x64 as the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, the "lscpu" terminal command was given:

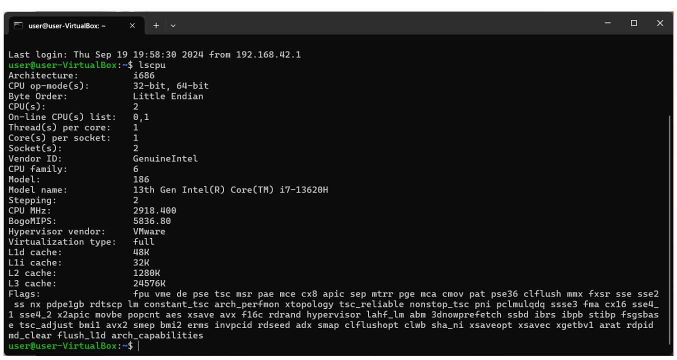


Fig. 1. Ubuntu CPU information

B. Virtual address layout in C

[1] Most machines typically address memory in the format of Figure 2 shown below. Here we are working on an x86 machine.

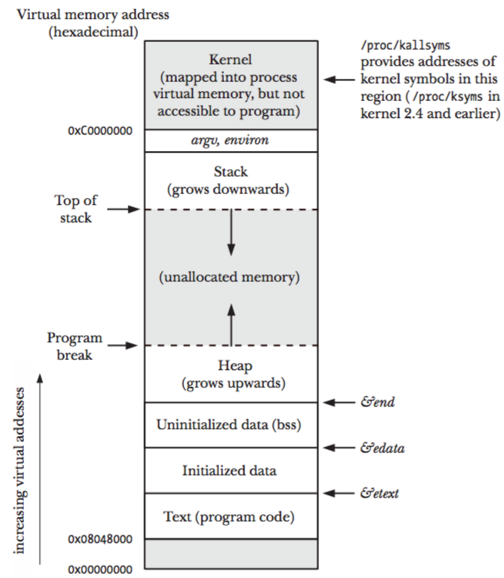


Fig. 2. Underlying memory mapping

The bottom of the stack contains "main()" functions and argument variables passed to it, along with other function parameters. The stack will grow downwards towards the lower address when local variables(or other code) are declared. The heap grows in the opposite direction and here runtime allocation of memory happens. The lowest memory assignments are done to program code and variables which are statically allotted.

C. Declaration types and memory addressing

[1] Based on the variable declaration and scope, memory is assigned differently by the compiler. Some of these are discussed below:

1) Storage class keywords: Auto is default is the default class and does not need explicit declaration. All local variables are auto and stored in the stack. Next is static which "0" is initialized by default and stored in the data section next to the program code. The values are not changed till the end of the program.

The extern keyword is similar to the global declaration variable and also stored in the data segment. Last is register, where the compiler uses CPU registers to compute but may/may not be stored in registers. Its value in memory can't be accessed.

2) Stack vs Heap: The stack grows from higher addresses to lower addresses and the memory is assigned at compile time. The bottom of the stack contains the frame pointer and return address. All the allocated space are towards the top and this includes the local variables or buffer.

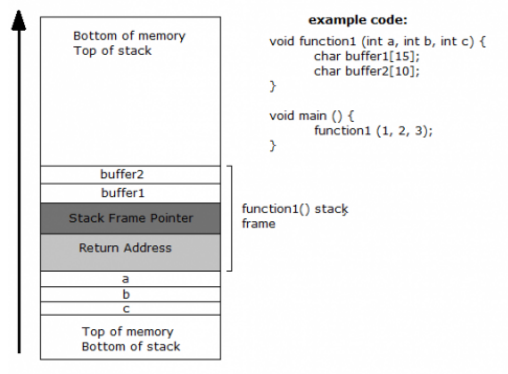


Fig. 3. Contents of a stack

On the contrary, the heap grows from lower to higher addresses and is used for dynamic memory allocation. The ”malloc()” function is one such example that uses the heap.

3) Initialized vs Uninitialized: During variable declaration, if a value is initialized, it gets stored in the data segment. This does not hold true for function local variables. All static, global, and extern-type stored values are here.

If no value is given, it is put in the ”.bss” in the data section. One thing is that if C by default puts some value to a variable, it is considered as initialized.

4) Global vs Local: If variables are declared outside a block or function, it is global and accessible to all. The C compiler puts these in the data section of memory. If a value is local, it is stored in the stack, and the scope is limited to that block. An exception is a static keyword for variables inside a function that is stored next to global ones.

D. Code Analysis and Verification

To compile the code, the following command was used for the GNU GCC compiler :

$$gcc\ -g\ address\_layout.c\ -o\ address\_layout \quad (1)$$

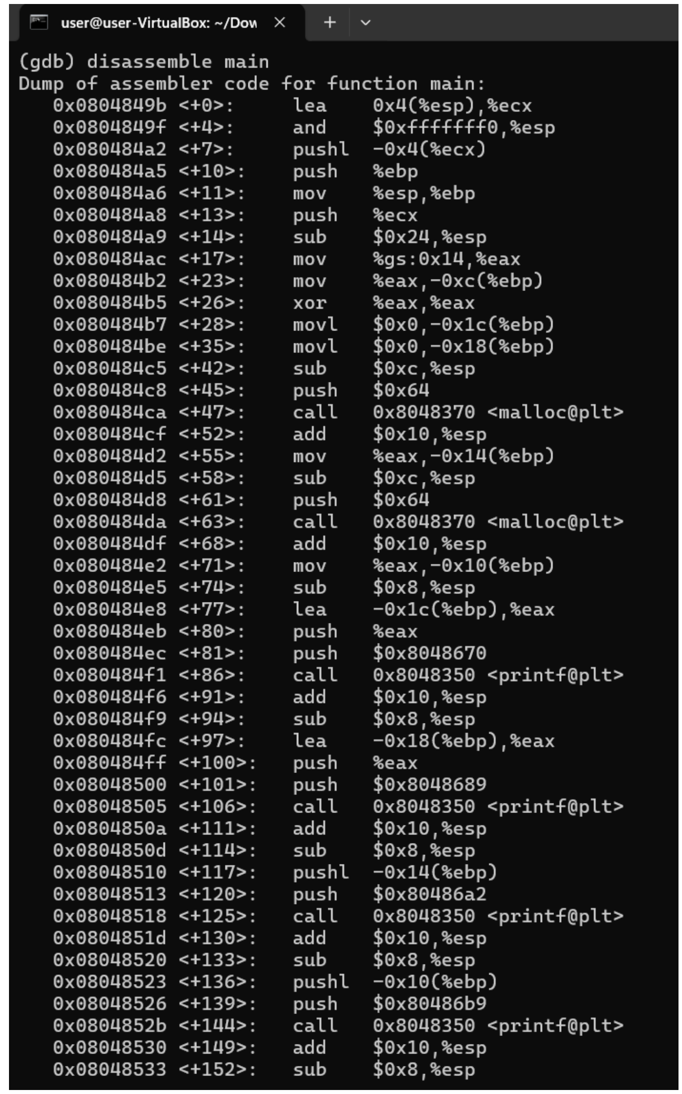
For generation of intermediate assembly code file :

$$gcc\ -g\ address\_layout.c\ -S\ address\_layout.s \quad (2)$$

To run the GDB against the executable with the automation script :

$$gdb\ address\_layout\ --batch\ --command = test.txt \quad (3)$$

- –batch: To exit the gdb session after processing the arguments. It prints to standard output.
- –command=file: It takes a file as an input in which gdb commands are present.
- test.txt is the text file for the experiment where the commands are written.



```
user@user-VirtualBox: ~/Dow
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804849b <+0>:     lea    0x4(%esp),%ecx
   0x0804849f <+4>:     and    $0xfffffff0,%esp
   0x080484a2 <+7>:     pushl  -0x4(%ecx)
   0x080484a5 <+10>:    push   %ebp
   0x080484a6 <+11>:    mov    %esp,%ebp
   0x080484a8 <+13>:    push   %ecx
   0x080484a9 <+14>:    sub    $0x24,%esp
   0x080484ac <+17>:    mov    %gs:0x14,%eax
   0x080484b2 <+23>:    mov    %eax,-0xc(%ebp)
   0x080484b5 <+26>:    xor    %eax,%eax
   0x080484b7 <+28>:    movl   $0x0,-0x1c(%ebp)
   0x080484be <+35>:    movl   $0x0,-0x18(%ebp)
   0x080484c5 <+42>:    sub    $0xc,%esp
   0x080484c8 <+45>:    push   $0x64
   0x080484ca <+47>:    call   0x8048370 <malloc@plt>
   0x080484cf <+52>:    add    $0x10,%esp
   0x080484d2 <+55>:    mov    %eax,-0x14(%ebp)
   0x080484d5 <+58>:    sub    $0xc,%esp
   0x080484d8 <+61>:    push   $0x64
   0x080484da <+63>:    call   0x8048370 <malloc@plt>
   0x080484df <+68>:    add    $0x10,%esp
   0x080484e2 <+71>:    mov    %eax,-0x10(%ebp)
   0x080484e5 <+74>:    sub    $0x8,%esp
   0x080484e8 <+77>:    lea    -0x1c(%ebp),%eax
   0x080484eb <+80>:    push   %eax
   0x080484ec <+81>:    push   $0x8048670
   0x080484f1 <+86>:    call   0x8048350 <printf@plt>
   0x080484f6 <+91>:    add    $0x10,%esp
   0x080484f9 <+94>:    sub    $0x8,%esp
   0x080484fc <+97>:    lea    -0x18(%ebp),%eax
   0x080484ff <+100>:   push   %eax
   0x08048500 <+101>:   push   $0x8048689
   0x08048505 <+106>:   call   0x8048350 <printf@plt>
   0x0804850a <+111>:   add    $0x10,%esp
   0x0804850d <+114>:   sub    $0x8,%esp
   0x08048510 <+117>:   pushl  -0x14(%ebp)
   0x08048513 <+120>:   push   $0x80486a2
   0x08048518 <+125>:   call   0x8048350 <printf@plt>
   0x0804851d <+130>:   add    $0x10,%esp
   0x08048520 <+133>:   sub    $0x8,%esp
   0x08048523 <+136>:   pushl  -0x10(%ebp)
   0x08048526 <+139>:   push   $0x80486b9
   0x0804852b <+144>:   call   0x8048350 <printf@plt>
   0x08048530 <+149>:   add    $0x10,%esp
   0x08048533 <+152>:   sub    $0x8,%esp
```

Fig. 4. disassembled main function

For viewing disassembled code :

$$(gdb)\ disassemble\ main \quad (4)$$

For viewing the source code :

$$(gdb)\ list\ 1,43 \quad (5)$$

[1]In the given ”address_layout.c” code, three types of variables are declared. Two of them are global variables, with and without initialization. The other one is a variable with the ”extern” keyword.

In the main function, four different types are present. The first set is local variables and static variables, followed by register-type and malloc functions. The malloc allocates the bytes mentioned in braces and returns a pointer. Here, a void pointer is assumed since no type-casting is done.

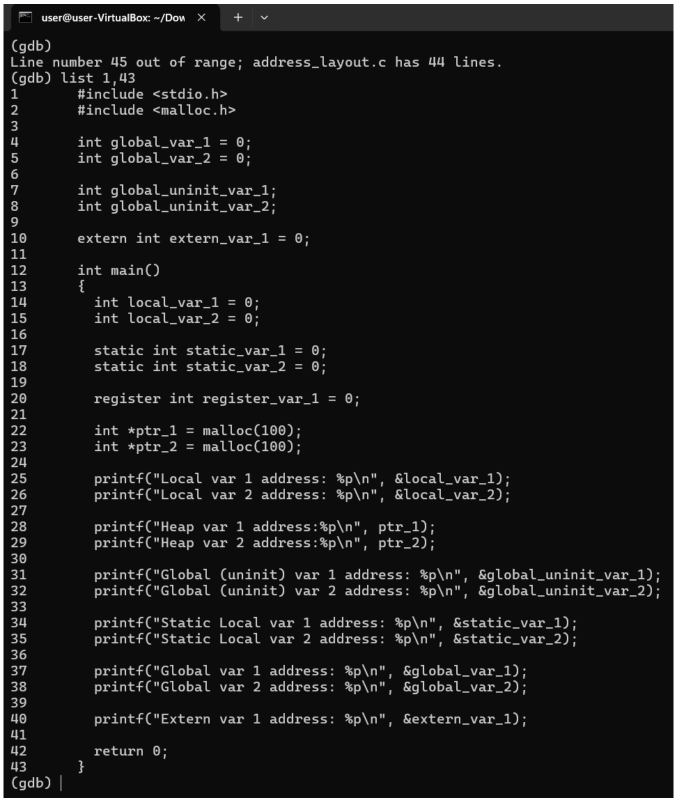


Fig. 5. C source with list command

III. Results

The following table captures the data derived from the console output when the code is run. The memory addresses are mapped to variable types and sorted from higher addresses to lower addresses.

TABLE I
VARIABLES DECLARATION TYPES AND THEIR MEMORY STORAGE (DESCENDING)

Sl. No.	Variable Type	Hex Address	Decimal Address
1	Local 2	0xbffff560	3221222752
2	Local 1	0xbffff55c	3221222748
3	Heap 2	0x804b070	134525040
4	Heap 1	0x804b008	134524936
5	Global (uninit) 2	0x804a040	134520896
6	Global (uninit) 1	0x804a03c	134520892
7	Static Local 2	0x804a038	134520888
8	Extern 1	0x804a034	134520884
9	Global 2	0x804a030	134520880
10	Global 1	0x804a02c	134520876
11	Static Local 1	0x804a024	134520868

IV. Discussions

A. GDB debugging with scripts

Using command (3), GDB ran and gave the console outputs 6 and 7. With the help of the script "test.txt" (See Appendix B), no user input was required. GDB takes the file as input and executes all the commands from top to bottom as if a user would do it manually.

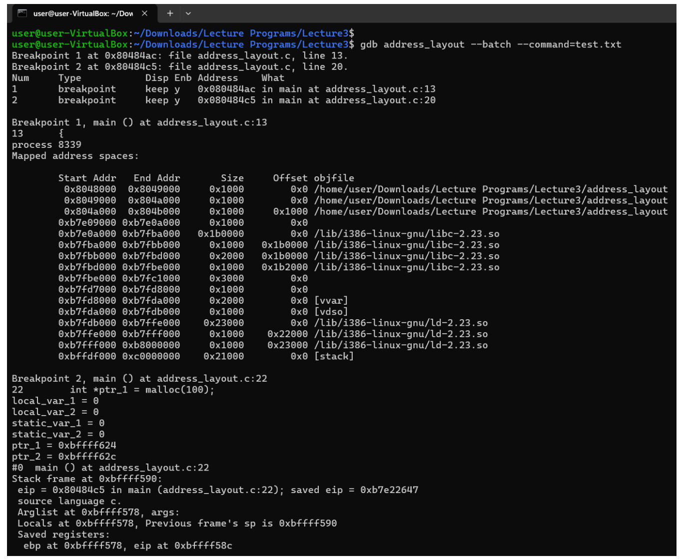


Fig. 6. GDB console output

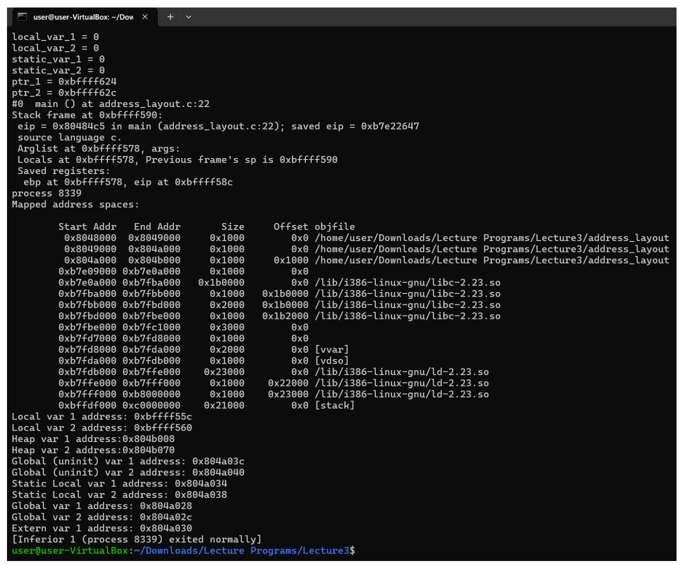


Fig. 7. GDB console output (contd.)

First, two breakpoints are added, one at the main and the other after declaration/initialization. We run the program after confirming those breakpoints. Once the first breakpoint hits, the process mapping with the memory is seen through :

$$info\ proc\ mappings \tag{6}$$

From Figure 6, all the sections of memory are displayed along with start/end addresses and size of regions. After the program is continued, the second breakpoint hits. Till now, all variables have been declared and initialized. The local variables, stack, and contents of the frame are checked :

$$\begin{aligned} &info\ locals \\ &bt \\ &info\ frame\ 0 \end{aligned} \tag{7}$$

By comparing the output of pointers and the second process mapping in Figure 7, the locations of variables in memory are accurate. The heap begins after the program code ends. The stack addresses are in the range as expected.

B. Variables and their storage in memory

From Table I above, we can easily see that local variables are stored in the stack at high-order addresses. ”1” and ”2” are just the variable numbers and ”2” is stored at a higher address than variable ”1”. The malloc function puts the variables in the heap and at lower addresses.

Below the heap, is the data segment where static, global variables are stored. Uninitialized values enjoy higher addresses than initialized ones. From the memory layout in Figure 2, we can verify the allotted memory order with the addresses found on the table. The only exception is the variable with register type as we are not allowed access to its memory by C.

C. Unexpected Results

Interestingly, some GDB commands do not work while taking input from a file. One of them is ”layout split” and when used, throws an error. Secondly, even after multiple runs post-compilation, the data segment values do not change their locations in memory and are static.

D. Limitations and Considerations

[1] The experiment was done on a Ubuntu VM and these address mapping changes if performed on another OS, or if executed on a different flavor. Where a variable is defined matters, for example, if static variables were declared in another function and not in the main. Also, different compilers may address memory differently than our GCC compiler.

Appendix A

For this experiment, the following C code has been utilized:

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

/*global declaration*/

int global_var_1 = 0;
int global_var_2 = 0;

/*uninitialized*/
int global_uninit_var_1;
int global_uninit_var_2;

 /*extern keyword used*/
extern int extern_var_1 = 0;

int main()
{
  /*local or auto type*/
  int local_var_1 = 0;
  int local_var_2 = 0;

  /* static */
  static int static_var_1 = 0;
  static int static_var_2 = 0;

  /*register*/
  register int register_var_1 = 0;

  /*100 bytes to int pointer*/
  int *ptr_1 = malloc(100);
  int *ptr_2 = malloc(100);

  /*printing*/
  printf("Local var 1 address: %p\n",\
  &local_var_1);
  printf("Local var 2 address: %p\n",\
  &local_var_2);

  printf("Heap var 1 address:%p\n", ptr_1);
  printf("Heap var 2 address:%p\n", ptr_2);

  printf("Global (uninit) var 1 address: %p\n",\
  &global_uninit_var_1);
  printf("Global (uninit) var 2 address: %p\n",\
  &global_uninit_var_2);

  printf("Static Local var 1 address: %p\n",\
  &static_var_1);
  printf("Static Local var 2 address: %p\n",\
  &static_var_2);

  printf("Global var 1 address: %p\n",\
  &global_var_1);
  printf("Global var 2 address: %p\n",\
  &global_var_2);

  printf("Extern var 1 address: %p\n",\
  &extern_var_1);

  return 0; }
```

Appendix B

The test.txt file with GDB commands:

```
b main
b 20
info b
run
info proc mappings
```

```
continue
info locals
bt
info frame 0
info proc mappings
continue
```

REFERENCES

[1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 3. pp. 1–3 (2024)