ENPM691 Homework Assignment 3

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 that the compiler uses for different variable declaration types and their probable cause. Debugger output will show how the compiler accesses these variables.**

***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.

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 understand by 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

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

Most machines typically address memory in the format of Figure 2 shown below. Here we are working on an x86 machine, but more or less the layout remains the same for all.

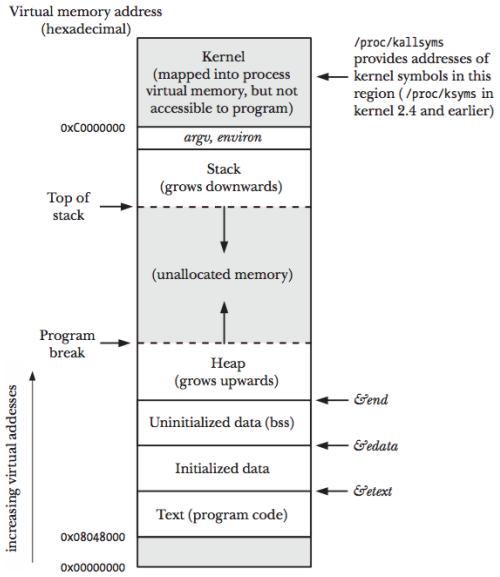

Fig. 2. Underlying memory mapping

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

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.

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Fig. 3. Storage classes in C

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.

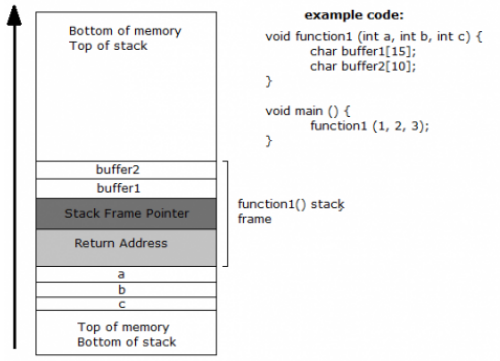

Fig. 4. Contents of a stack

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 compare the source code with the assembly code, gdb split was used :

$$\begin{aligned} &gdb\ address\_layout \\ &(gdb)\ layout\ split \end{aligned} \quad (3)$$

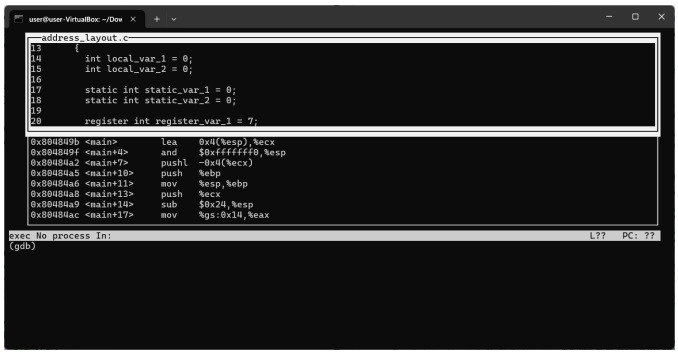

Fig. 5. gdb layout split

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.

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 Converted Address
1	Local 2	0xbf994aac	3214494380
2	Local 1	0xbf994aa8	3214494376
3	Heap 2	0x85d1070	140316784
4	Heap 1	0x85d1008	140316680
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

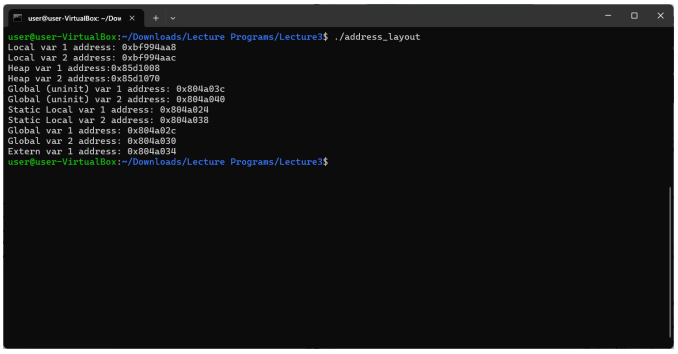

Fig. 6. Actual console output

IV. Discussions

A. 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, we have 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.

B. Unexpected Results

Interestingly, when ASLR was ”on” (/proc/sys/kernel/randomize_va_space) for the Ubuntu VM, the executable generated different memory locations for the stack and heap variables. The variables in the data segment did not change values.

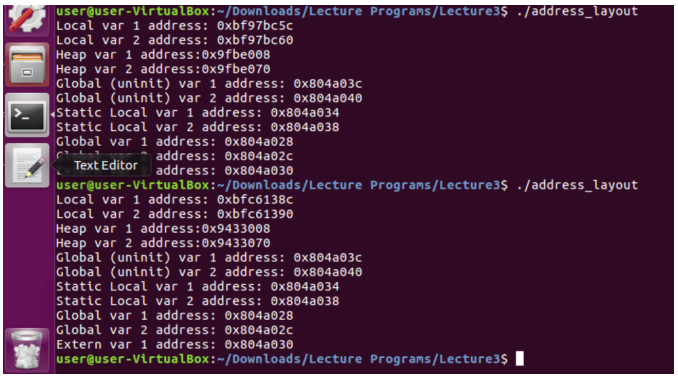

Fig. 7. Address change when ASLR was On

Another thing was though static variables were declared together, they were far apart in actual memory. Also, the compiler did not generate any specific assembly instruction for the static variable declaration.

C. Limitations and Considerations

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 main. Also, different compilers may address memory differently than our GCC compiler.

Appendix

C code

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;

int global_uninit_var_1; /*uninitialized*/
int global_uninit_var_2;

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

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

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

  register int register_var_1 = 0; /*register*/

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