Analyzing Memory Allocation Patterns of Compilers According to Various Elements of Variables

Jiwon Hwang *Cybersecurity*
University of Maryland, College Park
jhwang97@umd.edu

I. Introduction

A. Running Environment

Three different systems were used for comparison to analyze compiler behavior across different architectures and operating systems.

- System A
  - OS: Ubuntu 16.04.1 LTS
  - Architecture: i686 (32-bit)
  - Compiler: gcc v5.4.0
  - Debugger: gdb v7.11.1
- System B
  - OS: Ubuntu 24.04.1 LTS
  - Architecture: x86_64 (64-bit)
  - Compiler: gcc v13.2.0
  - Debugger: gdb v15.0.50
- System C
  - OS: macOS Ventura 13.6.5
  - Architecture: x86_64 (64-bit)
  - Compiler: gcc v14.0.3
  - Debugger: gdb v15.1

B. Definition

Memory allocation order notation: In this report, when variable A's memory allocation is before variable B's, it means that variable A has a smaller memory address. This relationship is indicated by A < B, where A is allocated at a lower memory address than B.

C. Purpose

The purpose of this report is to analyze how compilers allocate memory for variables based on different factors such as variable type, scope, initialization status, and declaration order. This analysis aims to provide insights into compiler optimization strategies and their implications for memory layout and security.

II. Methodology

A. Overview

The experimental approach involves analyzing simple C programs that declare variables with different characteristics and print their memory addresses. This methodology enables:

1) Exploring how the compiler allocates memory by examining program output
2) Understanding the rationale behind allocation patterns through debugging
3) Conducting systematic investigation by creating controlled test cases

B. Experimental Cases and Factors

To identify the factors influencing memory allocation patterns, the following test cases were designed:

- Case 1: Initialization Status (See Appendix A)
  Comparison between initialized and uninitialized variables
- Case 2: Variable Scope (See Appendix A)
  Local variables vs. global variables vs. static variables
- Case 3: Data Types (See Appendix B)
  Integer vs. long long vs. float types
- Case 4: Complex Data Types (See Appendix C)
  Structures, constants, and arrays
- Case 5: Function Parameters (See Appendix D)
  Parameter allocation patterns
- Case 6: Error Cases (See Appendix E)
  Stack and heap overflow scenarios

C. Analysis Methodology

Each test case was executed on all three systems, and the results were analyzed using debuggers to understand the underlying memory allocation mechanisms.

III. Results

A. Memory Allocation Patterns

The experimental results revealed that declaration order does not directly correlate with memory allocation order (See Tables 1-3). Instead, several key factors determine allocation patterns:

1) Case 1: Initialization Status (See Figure 1)
   Initialized variables < Uninitialized variables (in the data segment)
2) Case 2: Variable Scope (See Figure 1)
   Global < Static < Global(uninitialized) < Heap « Local
3) Case 3: Data Types (See Figures 2-3)
   Pattern remains consistent across different data types: Global < Static < Global(uninitialized) < Heap « Local
4) Case 4: Complex Data Types (See Figure 4)
   Global < Static < Global(uninitialized) < Heap « Array < Local < Constants < Structures
5) Case 5: Function Parameters (See Figure 5)
   Parameters are allocated in the stack frame: Global < Static < Global(uninitialized) < Heap variables = Heap parameters « Global parameters < Global(uninitialized) parameters < Local parameters < Static parameters < Local variables

B. System-Dependent Differences

Cross-platform analysis revealed that while general allocation patterns remain consistent, specific ordering within categories can vary by operating system. System C (macOS) showed reversed ordering for local variables compared to Systems A and B (Ubuntu), demonstrating platform-specific compiler optimizations (See Figure 6 and Table 6).

C. Overflow Analysis

Intentional buffer overflow experiments (See Appendix E) resulted in stack smashing detection and program termination, confirming the security implications of improper memory management (See Figure 10).

D. Tables

Table I
Memory Addresses from Appendix A (System A)

Variable	Address
Local var 2	0xbfdf251c
Local var 1	0xbfdf2518
...	
Heap var 2	0x9f86070
Heap var 1	0x9f86008
...	
Global (uninit) var 2	0x804a03c
Global (uninit) var 1	0x804a038
Static Local var 2	0x804a034
Static Local var 1	0x804a030
Global var 2	0x804a02c
Global var 1	0x804a028

Table II
Declaration Order

	Global	Local		Heap
		Non-Static	Static	
Initialized	1	3	5	4
Uninitialized	2	—	—	—

Table III
Memory Allocation Order

	Global	Local		Heap
		Non-Static	Static	
Initialized	1	5	3	4
Uninitialized	2	—	—	—

Table IV
MEMORY ADDRESSES FROM APPENDIX C (COMPLEX DATA TYPES)

Variable Type	Address
Struct var	0xbff4887c
Constant var	0xbff48860
Local var 2	0xbff4885c
Local var 1	0xbff48858
Array var	0xbff487e0
...	
Heap var 2	0x8ec4070
Heap var 1	0x8ec4008
...	
Global (uninit) var 2	0x804b04c
Global (uninit) var 1	0x804b048
Static Local var 2	0x804b044
Static Local var 1	0x804b040
Global var 2	0x804b03c
Global var 1	0x804b038

Table V
MEMORY ADDRESSES FROM APPENDIX D (FUNCTION PARAMETERS)

Variable/Parameter	Address
Local var 2	0xbfd415f0
Local var 1	0xbfd415ec
Static par 2	0xbfd415c4
Static par 1	0xbfd415c0
Local par 2	0xbfd415b4
Local par 1	0xbfd415b0
Global (uninit) par 2	0xbfd415ac
Global (uninit) par 1	0xbfd415a8
Global par 2	0xbfd415a4
Global par 1	0xbfd415a0
...	
Heap var=par 2	0x9212070
Heap var=par 1	0x9212008
...	
Global (uninit) var 2	0x804a040
Global (uninit) var 1	0x804a03c
Static Local var 2	0x804a038
Static Local var 1	0x804a034
Global var 2	0x804a030
Global var 1	0x804a02c

Table VI
MEMORY ADDRESSES FROM APPENDIX A IN SYSTEM C (MACOS)

Variable	Address
Local var 1	**0x7ff7b9920648**
Local var 2	**0x7ff7b9920644**
Local var 3	**0x7ff7b9920640**
...	
Heap var 2	0x600003bd8070
Heap var 1	0x600003bd8000
...	
Global (uninit) var 2	0x1065e700c
Global (uninit) var 1	0x1065e7008
Static Local var 2	0x1065e7014
Static Local var 1	0x1065e7010
Global var 2	0x1065e7004
Global var 1	0x1065e7000

IV. DISCUSSION

A. Memory Layout Analysis

The experimental results demonstrate that compiler memory allocation follows a systematic pattern based on variable characteristics rather than declaration order. According to the standard C program memory layout (See Figure 8), variables are organized into distinct memory regions:

- **Stack Area**
  - Local variables (excluding static)
  - Function parameters
  - Return addresses and local data
- **Heap Area**
  - Dynamically allocated variables
  - Variables allocated via malloc/calloc
- **Data Segment**
  - .data: Initialized global and static variables
  - .bss: Uninitialized global and static variables
- **Text Segment**
  - Program code and constants

The memory layout can be summarized as follows:

Table VII
MEMORY LAYOUT SUMMARY

Stack (High Addresses)
Structures, Constants, Local Variables, Arrays
Static, Local, Global Parameters
...
Heap (Medium Addresses)
Dynamically Allocated Variables, Parameters
...
Data Segment (Low Addresses)
.bss: Uninitialized Global and Static Variables
.data: Initialized Global and Static Variables
Text Segment (Lowest Addresses)
Program Code, String Literals

B. Compiler Optimization Strategies

The debugging analysis reveals the relationship between variable characteristics and compiler optimization strategies:

- **Stack Allocation**: Fast allocation and deallocation through simple pointer arithmetic. Optimal for variables with predictable, short lifetimes within function scope. The stack provides automatic memory management with minimal overhead.
- **Heap Allocation**: Dynamic allocation requiring system calls (malloc/free) and memory management overhead. Suitable for variables that must persist beyond function scope or have unpredictable lifetimes.

- **Data Segment Allocation**:
  - **.data segment**: Stores initialized global and static variables with values embedded in the executable
  - **.bss segment**: Efficiently handles uninitialized variables by reserving space without storing explicit zero values in the binary

C. Platform-Specific Behavior

The comparison between different systems reveals that while the general allocation pattern remains consistent, specific implementation details vary based on:

- Architecture (32-bit vs 64-bit)
- Operating system (Linux vs macOS)
- Compiler version and optimization settings

The reversed local variable ordering observed in System C (macOS) suggests different stack management strategies, possibly related to security features or performance optimizations specific to the platform.

D. Security Implications

Understanding memory allocation patterns is crucial for identifying potential security vulnerabilities:

- Buffer overflows can corrupt adjacent variables in predictable patterns
- Stack smashing attacks exploit the sequential nature of stack allocation
- Knowledge of memory layout helps in developing robust defensive programming practices

The overflow experiments (Appendix E) demonstrate the practical consequences of improper memory management, resulting in program termination and potential security exploitation.

V. Conclusion

This analysis demonstrates that compiler memory allocation follows systematic patterns based on variable scope, initialization status, and data types rather than declaration order. Understanding these patterns is essential for:

- Writing efficient code that aligns with compiler optimization strategies
- Identifying potential security vulnerabilities related to memory layout
- Developing defensive programming practices to prevent buffer overflows
- Understanding platform-specific behavior across different systems

The research reveals that modern compilers employ sophisticated strategies to optimize memory usage while maintaining program correctness. However, developers must remain aware of these patterns to avoid security pitfalls such as buffer overflows that can lead to serious vulnerabilities including those documented in CVE databases.

Future work could explore the impact of different compiler optimization levels and additional data types on memory allocation patterns, as well as investigate newer security features like stack canaries and address space layout randomization (ASLR).

VI. Appendices

A. Appendix A: address_layout.c

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

int global_var_1 = 0;
int global_var_2 = 0;

int global_uninit_var_1;
int global_uninit_var_2;

int main()
{
  int local_var_1 = 0;
  int local_var_2 = 0;
  int local_var_3 = 0;

  static int static_var_1 = 0;
  static int static_var_2 = 0;

  int *ptr_1 = malloc(100);
  int *ptr_2 = malloc(100);

  printf("Local var 1 address: %p\n", &
      local_var_1);
  printf("Local var 2 address: %p\n", &
      local_var_2);
  printf("Local var 3 address: %p\n", &
      local_var_3);

  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);

  return 0;
}
```

B. Appendix B: Data Type Variations

1) B1: address_layout_VarType_float.c:

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

float global_var_1 = 0;
float global_var_2 = 0;

float global_uninit_var_1;
float global_uninit_var_2;

float main()
{
  float local_var_1 = 0;
  float local_var_2 = 0;
  float local_var_3 = 0;

  static float static_var_1 = 0;
  static float static_var_2 = 0;

  float *ptr_1 = malloc(100);
  float *ptr_2 = malloc(100);

  printf("Local float var 1 address: %p\n", &
       local_var_1);
  printf("Local float var 2 address: %p\n", &
       local_var_2);
  printf("Local float var 3 address: %p\n", &
       local_var_3);

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

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

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

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

  return 0;
}
```

2) B2: address_layout_VarType_long_long.c:

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

long long global_var_1 = 0;
long long global_var_2 = 0;

long long global_uninit_var_1;
long long global_uninit_var_2;

long long main()
{
  long long local_var_1 = 0;
  long long local_var_2 = 0;
  long long local_var_3 = 0;

  static long long static_var_1 = 0;
  static long long static_var_2 = 0;

  long long *ptr_1 = malloc(100);
  long long *ptr_2 = malloc(100);

  printf("Local long long var 1 address: %p\n",
       &local_var_1);
  printf("Local long long var 2 address: %p\n",
       &local_var_2);
  printf("Local long long var 3 address: %p\n",
       &local_var_3);

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

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

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

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

  return 0;
}
```

C. Appendix C: address_layout_Union.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//#include <malloc.h>

int global_var_1 = 0;
int global_var_2 = 0;

int global_uninit_var_1;
int global_uninit_var_2;

// Define an enum to track the type of the
     variable
typedef enum {
    INT_TYPE,
    FLOAT_TYPE,
    CHAR_TYPE,
    DOUBLE_TYPE,
    LONG_LONG_TYPE,
    LONG_DOUBLE_TYPE
} VarType;

// Define a union to hold variables of
     different types
typedef union {
    int i;
    float f;
    char c;
    double d;
    long long int ll;
```

```
    long double ld;
} VarValue;

// Define a struct to hold both the type and
     value of the variable
typedef struct {
    VarType type;
    VarValue value;
} Var;

// Function prototypes
Var generateRandomVar();
void printVarDetails(Var var, void* address);
void printTableHeader();
size_t getVarSize(Var var);

int main() {
  int local_var_1 = 0;
  int local_var_2 = 0;

  static int static_var_1 = 0;
  static int static_var_2 = 0;

  int *ptr_1 = malloc(100);
  int *ptr_2 = malloc(100);

  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);

  // Define the size of the array
  const int ARRAY_SIZE = 5;
  Var varArray[ARRAY_SIZE];


  Var variable;
  printf("\n\n<< Additional Cases >\n");
  printf("Constant var 1 address: %p\n", &
       ARRAY_SIZE);
  printf("Array var 1 address: %p\n", varArray)
       ;
  printf("Union var 1 address: %p\n", &variable
       );



  srand(time(NULL)); // Seed the random number
       generator

    // Generate random variables and store them
         in the array
    for (int i = 0; i < ARRAY_SIZE; i++) {
        varArray[i] = generateRandomVar();
    }

    // Print table header
    printTableHeader();

    // Print details (type, address, and size)
         of each variable in tabular format
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printVarDetails(varArray[i], &varArray[
            i]);
    }

    return 0;
}

// Function to generate a random variable of
     random type
Var generateRandomVar() {
    Var var;
    int randomType = rand() % 6; // Randomly
         choose a type from 0 to 5

    switch (randomType) {
        case INT_TYPE:
            var.type = INT_TYPE;
            var.value.i = rand() % 100; //
                 Random int from 0 to 99
            break;
        case FLOAT_TYPE:
            var.type = FLOAT_TYPE;
            var.value.f = (float)rand() /
                 RAND_MAX * 100.0f; // Random
                 float from 0.0 to 100.0
            break;
        case CHAR_TYPE:
            var.type = CHAR_TYPE;
var.value.c = 'A' + (rand() % 26); // Random
     char from 'A' to 'Z'
            break;
        case DOUBLE_TYPE:
            var.type = DOUBLE_TYPE;
            var.value.d = (double)rand() /
                 RAND_MAX * 100.0; // Random
                 double from 0.0 to 100.0
            break;
        case LONG_LONG_TYPE:
            var.type = LONG_LONG_TYPE;
            var.value.ll = (long long int)rand
                 () * rand(); // Random long
                 long int
            break;
        case LONG_DOUBLE_TYPE:
            var.type = LONG_DOUBLE_TYPE;
            var.value.ld = (long double)rand()
                 / RAND_MAX * 100.0; // Random
                 long double from 0.0 to 100.0
            break;
    }

    return var;
}


// Function to print the table header
void printTableHeader() {
    printf("%-15s %-20s %-30s %-10s\n", "Type",
         "Address", "Value", "Size (bytes)");
    printf("
         ----------------------------------------------------------------
         n");
}

// Function to print the variable's details:
     type, address, value, and size in tabular
     format
void printVarDetails(Var var, void* address) {
    size_t size = getVarSize(var);

    switch (var.type) {
        case INT_TYPE:
```

```
            printf("%-15s %-20p %-30d %-10zu\n"
                , "int", address, var.value.i,
                size);
            break;
        case FLOAT_TYPE:
            printf("%-15s %-20p %-30f %-10zu\n"
                , "float", address, var.value.f
                , size);
            break;
        case CHAR_TYPE:
            printf("%-15s %-20p %-30c %-10zu\n"
                , "char", address, var.value.c,
                 size);
            break;
        case DOUBLE_TYPE:
            printf("%-15s %-20p %-301f %-10zu\n
                ", "double", address, var.value
                .d, size);
            break;
        case LONG_LONG_TYPE:
            printf("%-15s %-20p %-301ld %-10zu\
                n", "long long", address, var.
                value.ll, size);
            break;
        case LONG_DOUBLE_TYPE:
            printf("%-15s %-20p %-30Lf %-10zu\n
                ", "long double", address, var.
                value.ld, size);
            break;
        default:
            printf("Unknown type\n");
            break;
    }
}

// Function to get the size of the variable
    based on its type
size_t getVarSize(Var var) {
    switch (var.type) {
        case INT_TYPE:
            return sizeof(var.value.i);
        case FLOAT_TYPE:
            return sizeof(var.value.f);
        case CHAR_TYPE:
            return sizeof(var.value.c);
        case DOUBLE_TYPE:
            return sizeof(var.value.d);
        case LONG_LONG_TYPE:
            return sizeof(var.value.ll);
        case LONG_DOUBLE_TYPE:
            return sizeof(var.value.ld);
        default:
            return 0; // Unknown type
}
}
```

D. Appendix D: address_layout_Function.c

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

int global_var_1 = 0;
int global_var_2 = 0;

int global_uninit_var_1;
int global_uninit_var_2;

void parameterAnalysis(int global_par_1, int
    global_par_2, int global_uninit_par_1, int
    global_uninit_par_2, int local_par_1, int
    local_par_2, int *ptr_par_1, int *ptr_par_2
    , int static_par_1, int static_par_2);



int main()
{
  int local_var_1 = 0;
  int local_var_2 = 0;

  int *ptr_1 = malloc(100);
  int *ptr_2 = malloc(100);

  static int static_var_1 = 0;
  static int static_var_2 = 0;

  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);

  parameterAnalysis(global_var_1, global_var_2,
       global_uninit_var_1, global_uninit_var_2
      , local_var_1, local_var_2, ptr_1, ptr_2,
       static_var_1, static_var_2);

  return 0;
}


void parameterAnalysis(int global_par_1, int
    global_par_2, int global_uninit_par_1, int
    global_uninit_par_2, int local_par_1, int
    local_par_2, int *ptr_par_1, int *ptr_par_2
    , int static_par_1, int static_par_2){
  printf("\nParameters\n");
  //Print the variables as parameters
  printf("Local par 1 address: %p\n", &
      local_par_1);
  printf("Local par 2 address: %p\n", &
      local_par_2);

  printf("Heap par 1 address: %p\n", ptr_par_1)
      ;
  printf("Heap par 2 address: %p\n", ptr_par_2)
      ;

  printf("Global (uninit) par 1 address: %p\n",
       &global_uninit_par_1);
  printf("Global (uninit) par 2 address: %p\n",
       &global_uninit_par_2);

  printf("Static par 1 address: %p\n", &
      static_par_1);
  printf("Static par 2 address: %p\n", &
      static_par_2);

  printf("Global par 1 address: %p\n", &
```

```
        global_par_1);
    printf("Global par 2 address: %p\n", &
        global_par_2);
}
```

E. Appendix E: address_layout_Overflow.c

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

int global_var_1 = 0;
int global_var_2 = 0;

int global_uninit_var_1;
int global_uninit_var_2;

void stack_overflow_function() {
    char stack_buffer[10]; // Small buffer for
        stack overflow
    // Intentionally write beyond the stack
        buffer
    for (int i = 0; i < 20; i++) {
        stack_buffer[i] = 'A'; // This will
            cause a stack overflow
    }
}

int main() {
    int local_var_1 = 0;
    int local_var_2 = 0;
    int local_var_3 = 0;

    // Allocate memory on the heap
    int *ptr_1 = malloc(10 * sizeof(int)); //
        Small allocation
    int *ptr_2 = malloc(10 * sizeof(int)); //
        Another small allocation

    // Intentionally cause a heap overflow
    for (int i = 0; i <= 10; i++) { // Write 11
         integers into a 10-integer allocation
        ptr_1[i] = i; // This will cause a heap
             overflow
    }

    // Call a function that causes stack
        overflow
    stack_overflow_function();

    // Print addresses for debugging
    printf("Local var 1 address: %p\n", (void*)
        &local_var_1);
    printf("Local var 2 address: %p\n", (void*)
        &local_var_2);
    printf("Local var 3 address: %p\n", (void*)
        &local_var_3);
    printf("Heap var 1 address: %p\n", (void*)
        ptr_1);
    printf("Heap var 2 address: %p\n", (void*)
        ptr_2);
    printf("Global (uninit) var 1 address: %p\n
        ", (void*)&global_uninit_var_1);
    printf("Global (uninit) var 2 address: %p\n
        ", (void*)&global_uninit_var_2);
    printf("Global var 1 address: %p\n", (void
        *)&global_var_1);
    printf("Global var 2 address: %p\n", (void
        *)&global_var_2);

    free(ptr_1);
    free(ptr_2);

    return 0;
}
```

F. Figures

```
Local var 1 address: 0xbf85cd78
Local var 2 address: 0xbf85cd7c
Local var 3 address: 0xbf85cd80
Heap var 1 address:0x8e75008
Heap var 2 address:0x8e75070
Global (uninit) var 1 address: 0x804a038
Global (uninit) var 2 address: 0x804a03c
Static Local var 1 address: 0x804a030
Static Local var 2 address: 0x804a034
Global var 1 address: 0x804a028
Global var 2 address: 0x804a02c
```

Figure 1. Result of Appendix A in System A

```
user@user-VirtualBox:~/Assignments/3rd$ ./address_layout_VarType_float
Local float var 1 address: 0xbf9087a8
Local float var 2 address: 0xbf9087ac
Local float var 3 address: 0xbf9087b0
Heap float var 1 address:0x8858008
Heap float var 2 address:0x8858070
Global (uninit) float var 1 address: 0x804a038
Global (uninit) float var 2 address: 0x804a03c
Static Local float var 1 address: 0x804a030
Static Local float var 2 address: 0x804a034
Global float var 1 address: 0x804a028
Global float var 2 address: 0x804a02c
```

Figure 2. Result of Appendix B1 in System A

```
user@user-VirtualBox:~/Assignments/3rd$ ./address_layout_VarType_long_long
Local long long var 1 address: 0xbfc19780
Local long long var 2 address: 0xbfc19788
Local long long var 3 address: 0xbfc19790
Heap long long var 1 address:0x85e0008
Heap long long var 2 address:0x85e0070
Global (uninit) long long var 1 address: 0x804a050
Global (uninit) long long var 2 address: 0x804a058
Static Local long long var 1 address: 0x804a040
Static Local long long var 2 address: 0x804a048
Global long long var 1 address: 0x804a030
Global long long var 2 address: 0x804a038
```

Figure 3. Result of Appendix B2 in System A

```
/Share/Assignments/3rd # ./address_layout_Union
Local var 1 address: 0xbff48858
Local var 2 address: 0xbff4885c
Heap var 1 address:0x8ec4008
Heap var 2 address:0x8ec4070
Global (uninit) var 1 address: 0x804b048
Global (uninit) var 2 address: 0x804b04c
Static Local var 1 address: 0x804b040
Static Local var 2 address: 0x804b044
Global var 1 address: 0x804b038
Global var 2 address: 0x804b03c


<< Struct, Constant, Union >>
Struct var 1 address: 0xbff4887c
Constant var 1 address: 0xbff48860
Array (Init) var 1 address: 0xbff487e0
Type            Address             Value                       Size (bytes)
-------------------------------------------------------------------------------
long double     0xbff487e0          71.336520                   12
int             0xbff487f0          24                          4
long double     0xbff48808          88.310323                   12
long double     0xbff48810          54.686758                   12
char            0xbff48820          I                           1
```

Figure 4. Result of Appendix C in System A

```
Local var 1 address: 0xbfd415ec
Local var 2 address: 0xbfd415f0
Heap var 1 address:0x9212008
Heap var 2 address:0x9212070
Global (uninit) var 1 address: 0x804a03c
Global (uninit) var 2 address: 0x804a040
Static Local var 1 address: 0x804a034
Static Local var 2 address: 0x804a038
Global var 1 address: 0x804a02c
Global var 2 address: 0x804a030

Parameters
Local par 1 address: 0xbfd415b0
Local par 2 address: 0xbfd415b4
Heap par 1 address: 0x9212008
Heap par 2 address: 0x9212070
Global (uninit) par 1 address: 0xbfd415a8
Global (uninit) par 2 address: 0xbfd415ac
Static par 1 address: 0xbfd415c0
Static par 2 address: 0xbfd415c4
Global par 1 address: 0xbfd415a0
Global par 2 address: 0xbfd415a4
```

Figure 5. Result of Appendix D in System A

```
Local var 1 address: 0x7ff7b9920648
Local var 2 address: 0x7ff7b9920644
Local var 3 address: 0x7ff7b9920640
Heap var 1 address:0x600003bd8000
Heap var 2 address:0x600003bd8070
Global (uninit) var 1 address: 0x1065e7008
Global (uninit) var 2 address: 0x1065e700c
Static Local var 1 address: 0x1065e7010
Static Local var 2 address: 0x1065e7014
Global var 1 address: 0x1065e7000
Global var 2 address: 0x1065e7004
```

Figure 6. Result of Appendix A in System C

```
----------------------------------------
/Share/Assignments/3rd # ./address_layout
Local var 1 address: 0xbf9666c8
Local var 2 address: 0xbf9666cc
Local var 3 address: 0xbf9666d0
Heap var 1 address:0x84a3008
Heap var 2 address:0x84a3070
Global (uninit) var 1 address: 0x804a038
Global (uninit) var 2 address: 0x804a03c
Static Local var 1 address: 0x804a030
Static Local var 2 address: 0x804a034
Global var 1 address: 0x804a028
Global var 2 address: 0x804a02c
----------------------------------------
/Share/Assignments/3rd # ./address_layout
Local var 1 address: 0xbfbf5438
Local var 2 address: 0xbfbf543c
Local var 3 address: 0xbfbf5440
Heap var 1 address:0x9757008
Heap var 2 address:0x9757070
Global (uninit) var 1 address: 0x804a038
Global (uninit) var 2 address: 0x804a03c
Static Local var 1 address: 0x804a030
Static Local var 2 address: 0x804a034
Global var 1 address: 0x804a028
Global var 2 address: 0x804a02c
----------------------------------------
```

Figure 7. Result of Appendix A multiple tries in System A

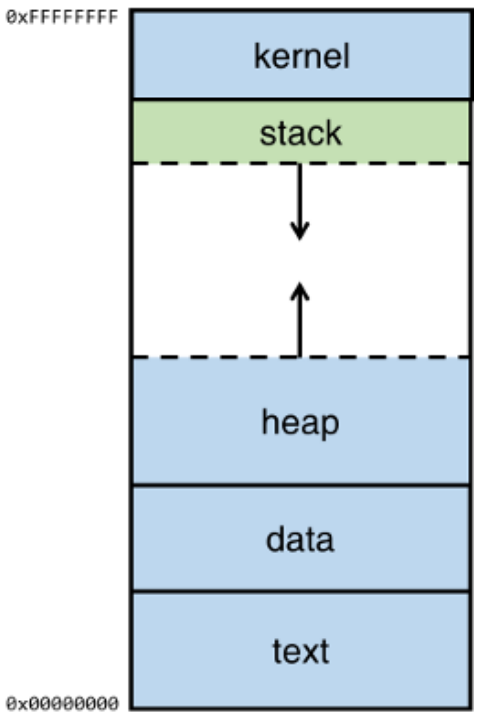

Figure 8. Program Memory Layout [2]

```
0x0804840b <+0>:     lea    0x4(%esp),%ecx
0x0804840f <+4>:     and    $0xfffffff0,%esp
0x08048412 <+7>:     pushl  -0x4(%ecx)
0x08048415 <+10>:    push   %ebp
0x08048416 <+11>:    mov    %esp,%ebp
0x08048418 <+13>:    push   %ecx
0x08048419 <+14>:    sub    $0x14,%esp
0x0804841c <+17>:    movl   $0x0,-0x10(%ebp)
0x08048423 <+24>:    sub    $0xc,%esp
0x08048426 <+27>:    push   $0x64
0x08048428 <+29>:    call   0x80482e0 <malloc@plt>
0x0804842d <+34>:    add    $0x10,%esp
0x08048430 <+37>:    mov    %eax,-0xc(%ebp)
0x08048433 <+40>:    mov    $0x0,%eax
0x08048438 <+45>:    mov    -0x4(%ebp),%ecx
0x0804843b <+48>:    leave
0x0804843c <+49>:    lea    -0x4(%ecx),%esp
0x0804843f <+52>:    ret
```

Figure 9. Debugging of simplified version of Appendix A in System A

```
/Share/Assignments/3rd # ./address_layout_Overflow
*** stack smashing detected ***: ./address_layout_Overflow terminated
[1]    12623 abort (core dumped)  ./address_layout_Overflow
```

Figure 10. Result of Appendix E in System A

REFERENCES

REFERENCES

[1] CVEdetails.com, "CVE security vulnerability database," Available: https://www.cvedetails.com/vulnerabilities-by-types.php

[2] C. Goedegebure, "A brief explanation of program memory and C++ memory management," Medium, Available: https://www.coengoedegebure.com/buffer-overflow-attacks-explained/