CYBR 437: Secure Coding

C Review-Part II

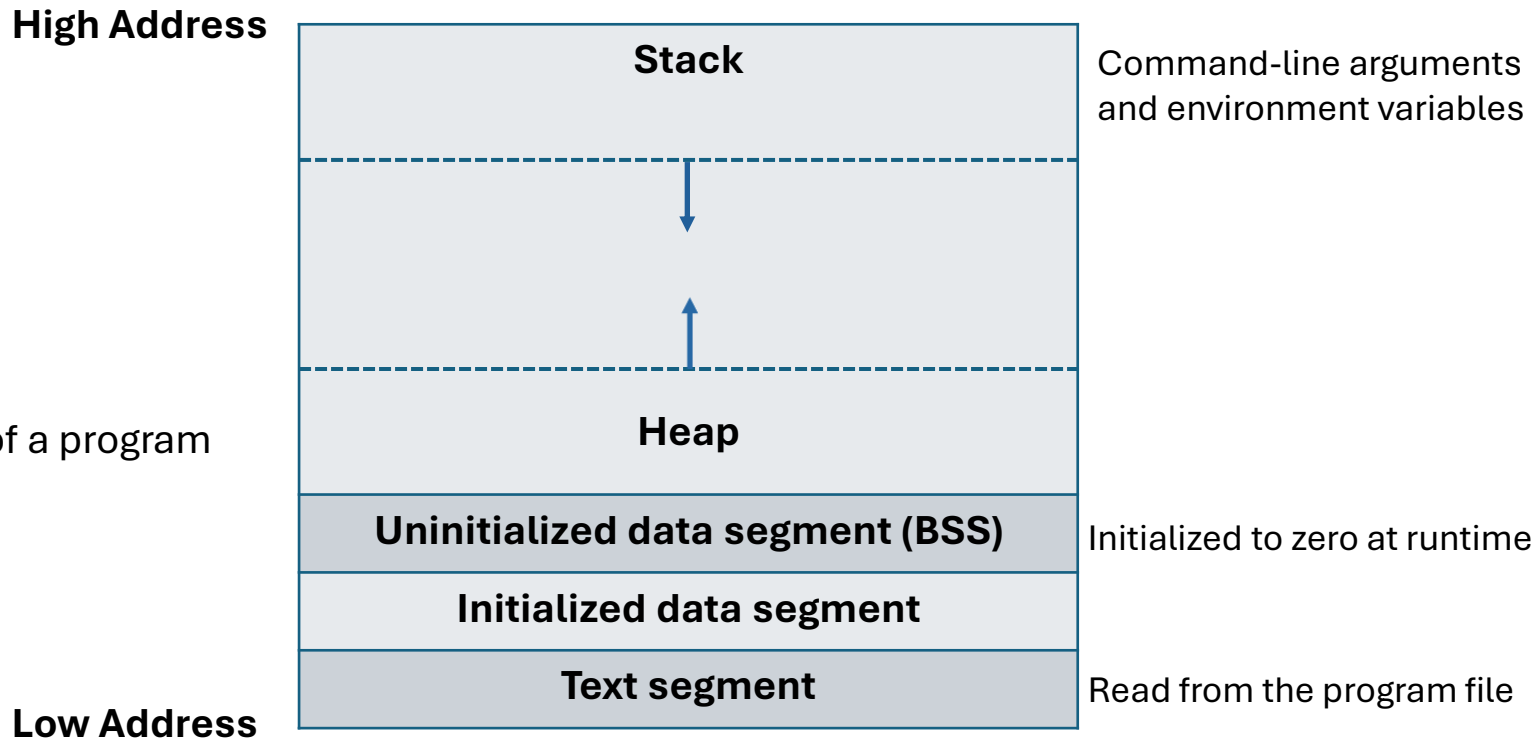
Instructor: Abinash Borah

Outline

- Memory Layout of C Programs
- Understanding the Stack and the Heap

Memory Layout of C Programs

- A typical memory representation of a C program consists of the following sections
  - Text/code segment
  - Initialized data segment
  - Uninitialized data segment (BSS)
  - Stack
  - Heap
- Called a process image – a captured state of a program
- Frozen image in time
- Process is in an executable state



Memory Layout of C Programs (contd.)

- Text/code segment
  - Stores the executable code of the program, such as functions and instructions
  - Usually read-only to prevent accidental modification during execution
  - Its size depends on the number of instructions and the complexity of the program
- Data segment
  - Stores global and static variables of the program
  - Variables in this segment retain their values throughout program execution
  - Its size depends on the number and type of global and static variables

  Two sections:

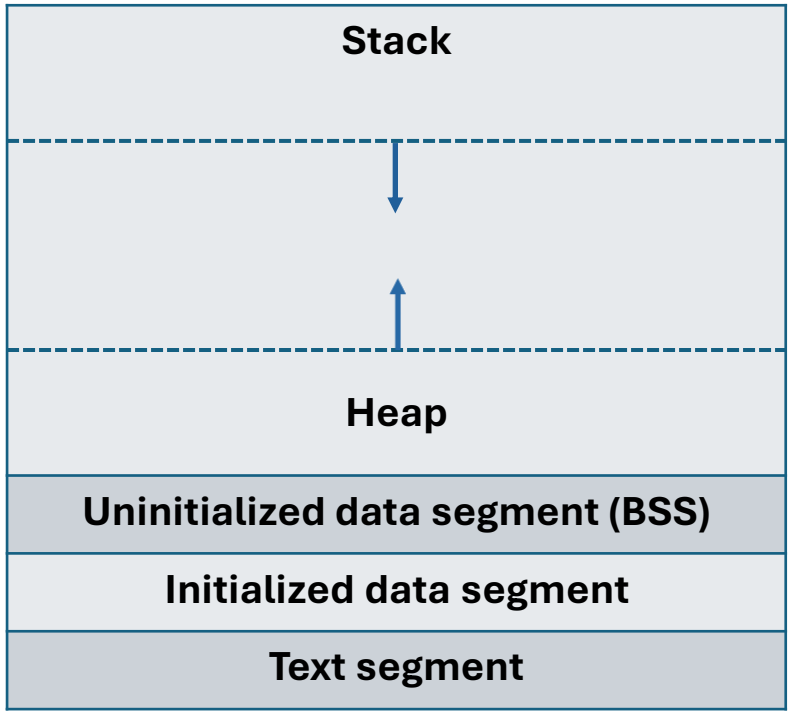
  - Initialized data segment – stores global and static variables that have been initialized by the programmer
  - Uninitialized data segment – stores global and static variables that are not initialized by the programmer

Memory Layout of C Programs (contd.)

- Heap
  - Heap is used for dynamic memory allocation
  - Starts at the end of the BSS segment and grows towards higher memory addresses
  - Memory in the heap is managed by using specific functions
- Stack
  - Stores local variables, function parameters, and return addresses for each function call
  - Each function call creates a stack frame in this segment
  - Function variables are pushed onto the stack when called
  - Function variables are popped off the stack when returns
  - Stack is at higher memory addresses and grows opposite to the heap

Where in memory will these variables be?

```
int x = 100;
int main ()
{
    int a = 2;
    float b = 2.5;
    static int y;
    int *ptr = (int *) malloc (2 * sizeof(int));
    ptr [0] = 10;
    ptr [1] = 15;
    free (ptr);
    return 1;
}
```



Why care about the stack and the heap?

- Understanding where items are located on the stack and in the heap creates the ability to compromise the code
- In a classic buffer overflow, we want to overflow the stack
- By understanding the stack, this overflow will allow us to replace a stack address with an address for a function we write
- Once we correctly replace the address, we can control the system

More on Function Stack Frames

- A function stack frame contains critical information for a specific function call:
  - **Return Address**: The memory address where the execution should return after the called function completes. It allows the program to resume execution from the point it left off.
  - **Parameters**: If the function accepts arguments, they are typically passed via the call stack or through registers. The stack frame stores these parameters for the function to utilize.
  - **Local Variables**: Each function has its own set of local variables. These variables are allocated within the stack frame and hold temporary data used during the function's execution.
  - **Saved Registers**: In some cases, certain registers need to be preserved before invoking a function to ensure the calling function’s state remains intact. The stack frame holds the values of these registers.
  - **Stack Frame Pointer**: A pointer, commonly referred to as the frame pointer or base pointer, keeps track of the current stack frame’s location. It helps access parameters, local variables, and other information within the stack frame.

How will the Stack change for this program?

```
#include<stdio.h>
#include<math.h>
void rms (int v)
{
    float r = sqrt(v);
    printf("The result is %f", r);
}
void ss (int a, int b, int c)
{
    int s = a * a + b * b + c * c;
    rms(s);
}
int main()
{
    int x = 3, y = 4, z = 5;
    ss(x, y, z);
    return 0;
}
```

Memory Management Functions

- void *malloc(size_t size);
  - The malloc()function allocates size bytes and returns a pointer to the allocated memory
  - The memory is not initialized
  - If size is zero, then malloc() returns either NULL or a unique pointer value that can later be passed to free()
- Void *realloc(void *ptr, size_t size);
  - The realloc() function shall deallocate the old object pointed to by ptr and return a pointer to a new object that has the size specified by size
  - The contents of the new object shall be the same as those of the old object before deallocation, up to the lesser of the new and old sizes
  - Any bytes in the new object beyond the size of the old object have indeterminate values
  - If the size of the space requested is zero, the behavior shall be implementation-defined: either a null pointer is returned, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object
  - If the space cannot be allocated, the object shall remain unchanged

Memory Management Functions

- void *calloc(size_t nmemb , size_t size);
  - The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory
  - The memory is set to zero
  - If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value that can later be successfully passed to free()
  - If the multiplication of nmemb and size would result in integer overflow, then calloc() returns an error
  - By contrast, an integer overflow would not be detected in the call malloc(nmemb * size); with the result that an incorrectly sized block of memory would be allocated
- void free(void *ptr);
  - The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc(), or realloc()
  - Otherwise, or if free(ptr) has already been called before, undefined behavior occurs
  - If ptr is NULL, no operation is performed