Project 3 Design Doc

Richard Nicholson, James Shedel, James Kuch, Mark Serrano, Tyler Smith

May 21, 2014

1 Design

1.1 Overview

In this project, we are to write tools that intercept memory calls from other programs. Here, we describe the correct behaviour for our set of tools that extend `malloc()` and `free()`.

1.2 slug_malloc()

This function will allocate memory by calling malloc(). It returns the address of allocated memory. In addition, it records the address, length, current timestamp, and location of the call in an internal data structure (2.2). If size is zero, this is not an error but should be reported on stderr as an unusual operation. If the input is excessively large (more than 128 MiB) then this function should report an error in stderr and terminate the program. The parameter WHERE is a string constant that records the filename and line number of the caller.

1.3 slug_free()

This function first checks that the addr is the start of a valid memory region that is currently allocated by looking through the internal data structures. If not, an errors is shown and the program terminated. If it is valid then free() should be called and the internal data structures updated to indicate that the address is no longer actively allocated.

1.4 slug_memstats()

This function traverses the internal data structures kept by slug_malloc() and slug_free() and prints out information about all current allocations. Each allocation report (2.2.2) should include the size of the allocation, a timestamp for when the allocation took place, the actual address of the allocation, and file and line number in the test program where the allocation happened. In addition, a summary of all allocations should be reported that includes the total number of allocations done, the number of current active allocations, the total amount of memory currently allocated, and the mean and standard deviation of sizes that have been allocated.

2 Implementation

2.1 Global Variables

- struct hashtable* ht
- int EXIT_STATUS

2.2 Internal Data Structure

We use a hash table as our internal data structure. It’s implementation is described in (2.2.1). To resolve hash collision, our team has decided to utilize linear probing.

2.2.1 Hash Table

- void init_hashtable()
    - initializes a hashtable with 2049 elements of the allocation struct (2.2.2).
    - upon program exit, all allocations made are shown in a table.
- void destroy_hashtable()
    - destroys the hash table and frees the memory that was used by it.
- void insert_into_hashtable(void* loc, char* where, time_t when, size_t size)
    - takes arguments, allocates memory for the allocation hash table data structure (2.2.2), and then inserts the data structure into the table at position: loc % (size of hash table).
- struct allocation* is_in_hashtable(void* loc)
    - takes memory location as argument, checks to see if the given location is located in the hash table by hashing the memory location and seeing if there is data at that position. if there is no data, then NULL is returned.

2.2.2 Allocation Struct

The allocation struct will have the following fields:

void* loc;	// the memory location returned by malloc
char* where;	// line number in file
time_t when;	// time allocated
size_t size;	// size of allocation
bool active;	// flag denoting if memory is freed or not

2.3 C Preprocessor Macros

2.3.1 Critical Macros

- #define malloc(s) slug_malloc((s), FILE_POS)
- #define free(s) slug_free((s), FILE_POS)

2.3.1 Helper Macros

- #define FILE_POS __FILE__ ":" INT2STRING(__LINE__)
- #define INT2STRING(i) FUNCTIONIZE(STRINGIZE,i)

- #define FUNCTIONIZE(a,b) a(b)
- #define STRINGIZE(a) #a

3 Testing Strategy

3.1 Correct Program

✓ perfect.c

3.2 Deallocate invalid address

✓ missedfree.c

3.3 Deallocate already freed region

✓ doublefree.c

3.4 Deallocating valid region

✓ notallocated.c

3.5 Input size >128 MiB

✓ largeinput.c

3.6 Allocating and exiting program

✓ notfreed.c

4 Results

4.1 Leak Detection

4.1.1 Always Detected

- After allocating and deallocating memory, trying to deallocate an invalid address is immediately detected (3.2).
- After allocating and deallocating memory, trying to deallocate an already freed region is immediately detected (3.3).
- After allocating and deallocating memory, trying to deallocate a valid region by passing in a pointer inside the region is immediately detected (3.4).
- Allocating memory larger than 128 MiB (3.5)

4.1.2 Sometimes Detected

- Allocating memory and then exiting triggers the leak detector and shows where the leak occurred (3.6).

4.2 Mean and Standard Deviation

Our program reports the mean and standard deviation of sizes that have been allocated.