Prof. Heonyoung Yeom

System Programming (001)

Lab #1 / Linker Lab

: Memtrace

Report

Dept. of Computer Science and Engineering
2021-13194
Jaejun Ko

2023.3.22

Lab #1 Report

Linker Lab : Memtrace

Introduction

In this lab session, we use **load/run time** library interpositioning to implement a dynamic memory allocation tracer. Our job is to trace all dynamic memory allocations/deallocations of a program without modifying source code, which occurred by calling the following functions, and to print out the name, the arguments, and the return value of each of them.

`void *malloc(size_t size)`

`malloc` allocates `size` bytes of memory on the process' heap and returns a pointer to it that can subsequently be used by the process to hold up to `size` bytes. The contents of the memory are undefined.

`void *calloc(size_t nmemb, size_t size)`

`calloc` allocates `nmemb*size` bytes of memory on the process' heap and returns a pointer to it that can subsequently be used by the process to hold up to `size` bytes. The contents of the memory are set to zero.

`void *realloc(void *ptr, size_t size)`

`realloc` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents are copied up to `min(size, old size)`, the rest is undefined.

`void free(void *ptr)`

`free` explicitly frees a previously allocated block of memory.

Part 1. Tracing Dynamic Memory Allocation

In Part 1, the basic part of the tracer will be implemented. Each function will conduct its task by logging what it did. This is the code, `memtrace.c` implemented to trace memory allocation/deallocation.

```
#define _GNU_SOURCE     // To use load/run interpositioning

#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <memlog.h>
#include <memlist.h>

static void *(*mallocp)(size_t size) = NULL;
static void (*freep)(void *ptr) = NULL;
static void *(*callocp)(size_t nmemb, size_t size);
static void *(*reallocp)(void *ptr, size_t size);

static unsigned long n_malloc  = 0;
static unsigned long n_calloc  = 0;
static unsigned long n_realloc = 0;
static unsigned long n_allocb  = 0;
static unsigned long n_freeb   = 0;
static item *list = NULL;
```

```
void *malloc(size_t size)
{
  char *error;
  void *ptr;

  if(!mallocp) {
    mallocp = dlsym(RTLD_NEXT, "malloc");
    if((error = dlerror()) != NULL) {
      fputs(error, stderr);
      exit(1);
    }
  }
  ptr = mallocp(size);
  LOG_MALLOC(size, ptr);
  n_malloc++;
  n_allocb += size;

  return ptr;
}

void *calloc(size_t nmemb, size_t size)
{
  char *error;
  void *ptr;

  if(!callocp) {
    callocp = dlsym(RTLD_NEXT, "calloc");
    if((error = dlerror()) != NULL) {
      fputs(error, stderr);
      exit(1);
    }
  }
  ptr = callocp(nmemb, size);
  LOG_CALLOC(nmemb, size, ptr);
  n_calloc++;
  n_allocb += nmemb * size;

  return ptr;
}

void *realloc(void *ptr, size_t size)
{
  char *error;
  void *ptr_res;

  if(!reallocp) {
    reallocp = dlsym(RTLD_NEXT, "realloc");
    if((error = dlerror()) != NULL) {
      fputs(error, stderr);
      exit(1);
    }
  }
  ptr_res = reallocp(ptr, size);
  LOG_REALLOC(ptr, size, ptr_res);
  n_realloc++;
  n_allocb += size;

  return ptr_res;
}

void free(void *ptr)
{
  char *error;

  if(!freep) {
    freep = dlsym(RTLD_NEXT, "free");
    if((error = dlerror()) != NULL) {
      fputs(error, stderr);
      exit(1);
    }
  }
  LOG_FREE(ptr);
  freep(ptr);
}
```

```
__attribute__((constructor))
void init(void)
{
  char *error;

  LOG_START();
  list = new_list();
}

__attribute__((destructor))
void fini(void)
{
  int avg_allocb = (n_malloc + n_realloc + n_calloc != 0) ? n_allocb /
(n_malloc+n_realloc+n_calloc) : 0;
  LOG_STATISTICS(n_allocb, avg_allocb, 0);

  LOG_STOP();
  free_list(list);
}
```

To implement load/run interpositioning, a pointer(e.g., `static void *(*mallocp)(size_t size)`...) is used for each function. First, each pointer is initialized to `NULL`. Whenever the function is called, the revised function checks if the pointer is `NULL`. If so, using the function `void *dlsym(void *handle, const char *symbol)` in library `<dlfcn.h>`, the original function becomes pointed by each pointer for the function by passing the symbol for the original function as `symbol`. If an error occurred, it prints an error message and exits. After checking, the revised function conducts its intended jobs and prints a log of what they did, by using macros defined in "`../utils/memlog.h`". By recording the number of each function called and the size of the memory block allocated, statistics for the program are produced. The following is the result code for each test case.

```
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part1$ make run test1
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x55b8f12202d0
[0003]          malloc( 32 ) = 0x55b8f12206e0
[0004]          malloc( 1 ) = 0x55b8f1220710
[0005]          free( 0x55b8f1220710 )
[0006]          free( 0x55b8f12206e0 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total          0
[0012]
[0013] Memory tracer stopped.
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part1$ make run test2
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x55e01af042d0
[0003]          free( 0x55e01af042d0 )
[0004]
[0005] Statistics
[0006]   allocated_total      1024
[0007]   allocated_avg        1024
[0008]   freed_total          0
[0009]
[0010] Memory tracer stopped.
```

```
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part1$ make run test3
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 11759 ) = 0x55f5ddc9c2d0
[0003]           calloc( 1 , 62125 ) = 0x55f5ddc9f0d0
[0004]           calloc( 1 , 29813 ) = 0x55f5ddcae390
[0005]           calloc( 1 , 37146 ) = 0x55f5ddcb5810
[0006]           calloc( 1 , 12629 ) = 0x55f5ddcbe940
[0007]           calloc( 1 , 61071 ) = 0x55f5ddcc1aa0
[0008]           calloc( 1 , 56778 ) = 0x55f5ddcd0940
[0009]           malloc( 18112 ) = 0x55f5ddcde720
[0010]           malloc( 3831 ) = 0x55f5ddce2df0
[0011]           calloc( 1 , 64981 ) = 0x55f5ddce3cf0
[0012]           free( 0x55f5ddce3cf0 )
[0013]           free( 0x55f5ddce2df0 )
[0014]           free( 0x55f5ddcde720 )
[0015]           free( 0x55f5ddcd0940 )
[0016]           free( 0x55f5ddcc1aa0 )
[0017]           free( 0x55f5ddcbe940 )
[0018]           free( 0x55f5ddcb5810 )
[0019]           free( 0x55f5ddcae390 )
[0020]           free( 0x55f5ddc9f0d0 )
[0021]           free( 0x55f5ddc9c2d0 )
[0022]
[0023] Statistics
[0024]   allocated_total      358245
[0025]   allocated_avg        35824
[0026]   freed_total          0
[0027]
[0028] Memory tracer stopped.
```

For each case, we can check that each function works correctly, and makes the log successfully.

Part 2. Tracing Unfreed Memory

Freeing after memory allocation is strongly recommended to avoid several problems, such as memory leaks. In Part 2, the tracer is required to trace non-deallocated memory, not only memory allocation/deallocation. Unless recording where memory is allocated, tracing non-deallocated memory is unavailable. So, to check where the memory is allocated and if it is freed is needed, the list defined in "`../utils/memlist.h`" will be used to record them. The following is `memtrace.c`, revised to trace non-deallocated memory. Redundant parts are omitted.

```
static item *list = NULL;
...

void *malloc(size_t size)
{
  char *error;
  void *ptr;
...
  ptr = mallocp(size);
  LOG_MALLOC(size, ptr);
  n_malloc++;
  n_allocb += size;
  alloc(list, ptr, size);
  return ptr;
}

void *calloc(size_t nmemb, size_t size)
{
  char *error;
  void *ptr;

...
  ptr = callocp(nmemb, size);
  LOG_CALLOC(nmemb, size, ptr);
  n_calloc++;
  n_allocb += nmemb * size;
  alloc(list, ptr, size);
  return ptr;
}

void *realloc(void *ptr, size_t size)
{
  char *error;
  void *ptr_res;

...
  n_freeb += dealloc(list, ptr)->size;

  ptr_res = reallocp(ptr, size);
  LOG_REALLOC(ptr, size, ptr_res);
  n_realloc++;
  n_allocb += size;
  alloc(list, ptr_res, size);
  return ptr_res;
}

void free(void *ptr)
{
  ...
  LOG_FREE(ptr);
  n_freeb += dealloc(list, ptr)->size;
  freep(ptr);
}

...
__attribute__((destructor))
void fini(void)
{
  int avg_allocb = (n_malloc + n_realloc + n_calloc != 0) ? n_allocb /
(n_malloc+n_realloc+n_calloc) : 0;
  LOG_STATISTICS(n_allocb, avg_allocb, n_freeb);

  item *i = list->next;
```

```
  int nonfreed_init = 0;
  do {
    if(i->cnt > 0) {
      if(!nonfreed_init) {
        LOG_NONFREED_START();
        nonfreed_init++;
      }
      LOG_BLOCK(i->ptr, i->size, i->cnt);
    }
    i = i->next;
  } while(i != NULL);

  LOG_STOP();
  free_list(list);
}
```

`item *alloc(item *list, void *ptr, size_t size)` adds information about a newly allocated block to the list, and `item *dealloc(item *list, void *ptr)` updates information on the freed block to the list and returns it. They are defined in "`../utils/memlog.c`". Each 'information' has four attributes, `void *ptr`, `size_t size`, `int cnt`, and `item *next`. By using these attributes, we can check if a pointer has been freed, and calculate how many bytes a program has freed. The following is the result of performing tracing for each test case.

```
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part2$ make run test1
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x564fcb47b2d0
[0003]          malloc( 32 ) = 0x564fcb47b710
[0004]          malloc( 1 ) = 0x564fcb47b770
[0005]          free( 0x564fcb47b770 )
[0006]          free( 0x564fcb47b710 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total          33
[0012]
[0013] Non-deallocated memory blocks
[0014]   block              size       ref cnt
[0015]   0x564fcb47b2d0     1024       1
[0016]
[0017] Memory tracer stopped.
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part2$ make run test2
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x56213af612d0
[0003]          free( 0x56213af612d0 )
[0004]
[0005] Statistics
[0006]   allocated_total      1024
[0007]   allocated_avg        1024
[0008]   freed_total          1024
[0009]
[0010] Memory tracer stopped.
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part2$ make run test3
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]          malloc( 26359 ) = 0x558e9ab632d0
[0003]          calloc( 1 , 46017 ) = 0x558e9ab69a00
[0004]          malloc( 1513 ) = 0x558e9ab74e00
[0005]          calloc( 1 , 40578 ) = 0x558e9ab75430
[0006]          malloc( 4067 ) = 0x558e9ab7f2f0
[0007]          malloc( 12143 ) = 0x558e9ab80310
[0008]          malloc( 51946 ) = 0x558e9ab832c0
[0009]          calloc( 1 , 53146 ) = 0x558e9ab8fdf0
[0010]          calloc( 1 , 10205 ) = 0x558e9ab9cdd0
[0011]          malloc( 34455 ) = 0x558e9ab9f5f0
[0012]          free( 0x558e9ab9f5f0 )
[0013]          free( 0x558e9ab9cdd0 )
[0014]          free( 0x558e9ab8fdf0 )
[0015]          free( 0x558e9ab832c0 )
[0016]          free( 0x558e9ab80310 )
[0017]          free( 0x558e9ab7f2f0 )
[0018]          free( 0x558e9ab75430 )
[0019]          free( 0x558e9ab74e00 )
[0020]          free( 0x558e9ab69a00 )
[0021]          free( 0x558e9ab632d0 )
[0022]
[0023] Statistics
[0024]   allocated_total      280429
[0025]   allocated_avg        28042
[0026]   freed_total          280429
[0027]
[0028] Memory tracer stopped.
```

For each test case, we can check that the tracer traced non-deallocated memory blocks successfully and made the log correctly. If there does not exist non-deallocated memory blocks, the tracer does not produce any log about them, as it should.

Part 3. Detecting and Ignoring Illegal Deallocations

In memory deallocation, **illegal free** or **double free** can occur. Illegal free means deallocating a memory block not allocated, and double free means deallocating a memory block freed already. Since they are serious errors, illegal deallocation will be detected and ignored in Part 3.

In the case of illegal free, the memory block not allocated is not in the list. Thus, the tracer failed to find the memory block in the list during deallocation, which means that the program is committing an illegal free. Using the function `item *find(item *list, void *ptr)`, the tracer can verify that the memory block to be deallocated is in the list, and if it exists, it can obtain information about the memory block. (If the function returns `NULL`, the memory block is not in the list.)

If a memory block that is in the list but whose reference count is 0 is going to be deallocated, it may be said that a double free has occurred. i.e., if the `cnt` of the information retrieved by `find(item *list, void *ptr)` is 0, the tracer will determine that a double free has occurred and ignore deallocation. The following code is the final version of our implementation. The overlapped codes with Part 2 are omitted.

```
...

void free(void *ptr)
{
  char *error;
...

  LOG_FREE(ptr);
  item* find_res = find(list, ptr);
  if(find_res == NULL){
    LOG_ILL_FREE();
  } else if(find_res->cnt == 0) {
    LOG_DOUBLE_FREE();
  }
  else{
    n_freeb += dealloc(list, ptr)->size;
    freep(ptr);
  }
}
```

The following is the result of applying the tracer to test cases.

```
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part3$ make run test4
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x559dacbed2d0
[0003]          free( 0x559dacbed2d0 )
[0004]          free( 0x559dacbed2d0 )
[0005]     *** DOUBLE_FREE  *** (ignoring)
[0006]          free( 0x1706e90 )
[0007]     *** ILLEGAL_FREE *** (ignoring)
[0008]
[0009] Statistics
[0010]   allocated_total      1024
[0011]   allocated_avg        1024
[0012]   freed_total          1024
[0013]
[0014] Memory tracer stopped.
jaejun@jaejun-ZenBook-UX433FN-UX433FN:~/Desktop/System Programming/HW/HW1/linklab/handout/part3$ make run test5
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlist.c ../utils/memlog.c -ldl
[0001] Memory tracer started.
[0002]          malloc( 10 ) = 0x56356cda12d0
[0003]          realloc( 0x56356cda12d0 , 100 ) = 0x56356cda1320
[0004]          realloc( 0x56356cda1320 , 1000 ) = 0x56356cda13c0
[0005]          realloc( 0x56356cda13c0 , 10000 ) = 0x56356cda17e0
[0006]          realloc( 0x56356cda17e0 , 100000 ) = 0x56356cda3f30
[0007]          free( 0x56356cda3f30 )
[0008]
[0009] Statistics
[0010]   allocated_total      111110
[0011]   allocated_avg        22222
[0012]   freed_total          111110
[0013]
[0014] Memory tracer stopped.
```

For test 4, we can check that the tracer detected and ignored all the illegal deallocations successfully. If there does not exist any illegal deallocations, it works normally, as we can see in the case of test 5.

Discussion

In this lab session, it was hard to understand how to use `memlist.h` and `memlist.c` in the implementation of the tracer. The question was why `dealloc` does not remove the entry about the freed memory block in Part 2. The answer was in Part 3, the tracer can identify double free and illegal free, by not removing the entry of the memory block already deallocated. If the information had been erased, the tracer would not have been able to distinguish whether the illegal deallocation was double free or illegal free.

Until now, there were some questions about whether there are any ways to add or remove certain functions in the library function or to print a log for errors. Many parts could be solved using exception handling or conditional statements, but there were some cases where there was no function that exactly fits the desired implementation, and there has been the desire to modify some functions appropriately to satisfy the implementation. From the lecture and this lab session, library interpositioning could be used to satisfy such needs. Moreover, it was able to get direct inspiration that library interpositioning could be applied on the security side.

Conclusion

In this lab session, we were able to implement a dynamic memory allocation/deallocation tracer by using the **load/run time** library interpositioning. The additional capability to detect unfreed memory blocks and ignore errors that occurred by conducting illegal deallocation is added to the tracer. it worked for every test case by part.