|  |
| --- |
|  |
| **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.

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.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

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

1. **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.

Text

Description automatically generated

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.

1. **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.

Text

Description automatically generated

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.