

Homework 1: Trace Memory Leaks

Nombre: Ing. Luis Carlos Alvarez Mata
Nombre: Ing. Verny Moreles Soto

Carne: 200902144
Carne: 201116212

Las fallas de caché y las pérdidas de memoria son un problema común en el desarrollo de software, apenas se notan cuando se trabaja en computadoras personales, sin embargo, cuando se ejecuta un programa con fugas en el sistema incorporado, las cosas generalmente salen mal. Los sistemas embebidos comunes tienen una cantidad limitada de memoria y el uso correcto es crítico para evitar el bloqueo del sistema en los primeros 5 minutos de operaciones. Con esta tarea, el estudiante comenzará a experimentar con herramientas comunes de análisis de memoria y comprenderá cómo rastrearlas efectivamente dentro de un código en ejecución.

1. Implementación

1.1. Memcheck

Este código se encarga de ejecutar un archivo binario y enlaza el programa libmemchck. Para esto se le solicita al usuario la dirección del ejecutable. Donde por medio de la función `execve` se realiza el enlace dinámico.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <getopt.h>

int main (int argc, char **argv) {

    int i = 0;
    int opt = -1;
    int memoryLeakFlag = 0;
    int infoAuthorFlag = 0;
    int infoExecutionAppFlag = 0;

    char *pathFile;
    char *const args[] = {pathFile, NULL};
    char *const envp[] = {"LD_PRELOAD=./libmemcheck.so", NULL};

    char usageMessage[] = "\n# USAGE MODE: \n"
        "./memcheck [ -p ./PROGRAM ] [-h] [-a] \n"
        "-a displays the information of the author of the program. \n"
        "-h displays the usage message to let the user know how to execute the application. \n"
        "-p PROGRAM specifies the path to the program binary that will be analyzed \n"
        "Example:\n"
        "./memcheck -p path/case4\n";

    char authorsInfo[] = "\n# AUTHORS INFORMATION\n"
        "# Tecnológico de Costa Rica (www.tec.ac.cr)\n"
        "# Course: MP-6171 High Performance Embedded Systems\n"
        "# Developers Name: Verny Morales and Luis Carlos Alvarez\n"
        "# Developers email: verny.morales@gmail.com and lcarn03@gmail.com\n";
```

```
while ((opt = getopt(argc, argv, "ahp")) != -1) {
    i++;
    switch(opt) {
        case 'p':
            pathFile = argv[i+1];
            memoryLeakFlag = 1;
            i++;
            break;
        case 'a':
            infoAuhtorFlag = 1;
            break;
        case 'h':
            infoExcutionAppFlag = 1;
            break;
        default:
            printf("Error..");
    }
}

if (infoAuhtorFlag == 1){
    printf("%s", authorsInfo);
}

if (infoExcutionAppFlag == 1){
    printf("%s", usageMessage);
}

if (memoryLeakFlag == 1){
    printf("\n#Analyzing the amount of memory leaks for the program: %s \n", pathFile);
    execve(pathFile, args, envp);
}
}
```

1.2. Libmenchck

Se realizo la reimplementación de las funciones de sistema malloc y free por medio de la llamada del sistema sbrk. Donde por medio de esta llamada se pueden manipular la pila y heap para los procesos crear el espacio necesario para los procesos.

Se creo un struct donde almacenaremos, el tamaño del bloque, si está o no libre, y cuál es el bloque siguiente. Esto nos ayudará a conocer el tamaño reservado para cada puntero, con lo cual al realizar un free vamos a saber cuánto espacio debe ser liberado. Los struct serán almacenados en una lista enlazada,

```
struct block_meta {
    size_t size;
    struct block_meta *next;
    int free;
    int magic;
};
```

Al realizar un malloc, es una buen practica reusar el espacio libre si fuera posible, por lo tanto, se recorrerá la lista enlazada buscando un bloque libre con tamaño suficiente. Si se encuentra un bloque que funcione, se debe solicitar sl sistema operativo el espacio usando sbrk y se agrega a la lista.

Si la lista enlazada es nula, se solicita el espacio el sistema operativo, si no es nulo se busca reutilizar algún bloque. Se devuelve block+1 debido a que se desea devolver un puntero a la región siguiente. Al mismo tiempo

incrementamos la variable mallocNumber para contabilizar la cantidad de malloc realizados.

```
void *malloc(size_t size) {  
  
    struct block_meta *block;  
  
    if (size <= 0) {  
        return NULL;  
    }  
  
    if (!global_base) { // First call.  
        block = request_space(NULL, size);  
    }  
    if (!block) {  
        return NULL;  
    }  
    global_base = block;  
    } else {  
        struct block_meta *last = global_base;  
        block = find_free_block(&last, size);  
        if (!block) { // Failed to find free block.  
            block = request_space(last, size);  
            if (!block) {  
                return NULL;  
            }  
        } else { // Found free block  
            block->free = 0;  
            block->magic = 0x77777777;  
        }  
    }  
  
    mallocNumber++;  
    return(block+1);  
}
```

Inicialmente se verifica que el puntero que está entrando como parámetro no sea nulo. Free no debería llamarse con cualquier dirección arbitraria o bloques que ya hayan sido liberados, por lo tanto, se utiliza la función assert que esas cosas no pasen. Y cambiamos la bandera a que ahora es un bloque free. Al finalizar el método se incrementa la variable freeNumber para contabilizar la cantidad de llamadas.

```
void free(void *ptr) {  
    if (!ptr) {  
        return;  
    }  
  
    struct block_meta* block_ptr = get_block_ptr(ptr);  
    assert(block_ptr->free == 0);  
    assert(block_ptr->magic == 0x77777777 || block_ptr->magic == 0x12345678);  
    block_ptr->free = 1;  
    block_ptr->magic = 0x55555555;  
  
    freeNumber++;  
}
```

Para imprimir la cantidad de malloc y free realizados, se crea la función writeMemoryLeaks la cual imprime un mensaje con la cantidad de estos y la cantidad de memory leaks que existen en el sistema. Este método se ejecuta al terminar el programa por medio del destructor. Cabe resaltar que se debe eliminar un malloc, debido a que este se

ejecuta al iniciar el programa donde esta asignando la memoria para este.

```
void __attribute__((destructor)) writeMemoryleaks();

void writeMemoryleaks(){
    printf("\n#Analysis finished! \nMemory allocations: %d"
           "\nMemory free: %d \nTotal memory leaks found: %d\n\n"
           ,mallocNumber-1, freeNumber, mallocNumber-1-freeNumber);
}
```

1.3. Autotools

Para poder compilar y generar los binarios respectivos para memcheck.c, y además generar el .so para libmemcheck, se va a utilizar Autotools. A continuación se muestra el archivo configure.ac, y cada dependencia.

```
AC_INIT([memcheck libmemcheck], [1.0], [sercr0388@address])
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
AM_PROG_AR
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([Makefile src/Makefile lib/Makefile])
AC_PROG_RANLIB
LT_INIT
AC_OUTPUT
```

Como se puede observar se agregan los respectivos Makefile, tanto para el src como para lib. Así como algunas librerías necesarias para generar el .so . A continuación se muestran los archivos de Makefile para el memcheck.c como para libmemcheck.c.

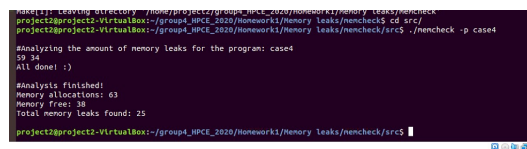
```
bin_PROGRAMS = memcheck
memcheck_SOURCES = memcheck.c
```

```
lib_LTLIBRARIES = libmemcheck.la
libmemcheck_la_SOURCES = libmemcheck.c
libmemcheck_la_CFLAGS = -fPIC -ldl -shared
noinst_LIBRARIES = libmemcheck.a
```

Como se observa se utilizan dos Makefile por separado uno para cada código c. Es importante denotar la diferencia para el Makefile de la librería, en vez de generar un binario, se genera un conjunto de archivos los cuales se utilizan para el comando LD PRELOAD, indispensable para correr el código de manera automática.

2. Resultados

Como caso de prueba se ejecuto el archivo binario llamada case4, brindado por el profesor del curso. Donde este binario realiza malloc y free de forma aleatoria e imprime la cantidad de veces que lo realiza.



```
case4: Leaving directory: /home/project2/group4_MPCE_2020/homework1/memory_leaks/memcheck
project2@project2-VirtualBox:~/group4_MPCE_2020/homework1/memory_leaks/memcheck$ cd src/
project2@project2-VirtualBox:~/group4_MPCE_2020/homework1/memory_leaks/memcheck/src$ ./memcheck -p case4
#Analyzing the amount of memory leaks for the program: case4
59.84
All done! :)
#Analysis finished!
Memory allocations: 63
Memory free: 30
Total memory leaks found: 25
project2@project2-VirtualBox:~/group4_MPCE_2020/homework1/memory_leaks/memcheck/src$
```

Figura 1: Ejecucion del binario case4

En la figura 1 se muestra el resultado, donde se encuentran un total de 25 memory leaks, data que es correcto si se restan la cantidad de malloc y free que el programa imprime. Sin embargo, el programa como tal por efecto de cargarse a memoria ejecuta ciertos llamados a esta que no son detectados por el programa case4, siendo estos 5 llamados en cada caso lo cual genera un offset en la medición.

Referencias

- [1] “attribute((constructor)) and attribute((destructor)) syntaxes in C in tutorials point?” <https://www.tutorialspoint.com/attribute-and-attribute-destructor-syntaxes-in-c-in-tutorials-point> (accessed Jul. 19, 2020).
- [2] “Debug: Understanding Valgrind’s messages — EPITECH 2022 - Technical Documentation 1.3.38 documentation.” <http://docs.epitech2022.eu/en/latest/valgrind.html> (accessed Jul. 19, 2020).
- [3] “Malloc tutorial.” <https://danluu.com/malloc-tutorial/> (accessed Jul. 19, 2020).
- [4] Tuxthink, “Linux World: Using execve,” Linux World. <https://tuxthink.blogspot.com/2012/06/using-execve.html> (accessed Jul. 19, 2020).