Using Harris, we will demonstrate and discuss Valgrind to detect memory-related errors.

Darien Tan Shi Feng, Ole Yannick Schlüter

June 20, 2023

Abstract

This project uses the Harris platform and the dynamic analysis tool Valgrind to identify memory-related issues in the C programming language. If not correctly controlled, memory allocation and deallocation and free can result in errors. The double-free is a frequent mistake where the same memory block is deallocated more than once, leading to unpredictable behavior. Such errors can be found using Valgrind's Memcheck tool, which intercepts function calls and examines memory activities. In this project, we test the detection of double-free, no free, reading outside of the malloc'd block and accessing outside the array length errors using Memcheck using flawed code, then we examine the generated reports. We also review how to use Valgrind to find and correct memory-related issues, demonstrating its value in enhancing program reliability. Through this research, we emphasize the value of effective memory management and the function that dynamic analysis tools like Valgrind play in ensuring memory safety in C.

1 Introduction

The *C* programming language offers different ways to store variables in memory. One option is dynamic memory allocation. The advantage of dynamic memory allocation is, that it allows the program to allocate memory blocks at runtime. This is needed when you do not know the length of the required memory before running the application eg. when reading a file from storage. The disadvantage is, that it is the responsibility of the programmer to manage these allocated blocks. Memory blocks are allocated using the functions `malloc`, `calloc` and `realloc`. These functions return a pointer to the allocated memory block. If they are no longer needed, the memory is deallocated using `free`, to make the memory available again. The misuse of dynamic memory is a common source of bugs.

One such misuse is known as *double free* and appears in the `simple_double_free()` function in lines four and five (see figure1). It is the repeated deallocation of the same memory block, that has been allocated before. The *C* standard states that the behavior of this is undefined. Undefined behavior should be avoided. Even if the program runs without issues on your machine, it is not guaranteed to work on others or can leave the memory in a

corrupted state. This can even become a security risk[1].
One might think, that this error is very unlikely to happen because no one will write code as in the example. However, this is a toy example, most software has a lot more than a single function and can become quite large. Additionally, pointers with different names can point to the same memory block or a free can depend on conditions, making the situation more complex (see `complex_double_free()` in figure1). A recent example (February 2, 2023) where a double free appeared is *OpenSSH*, a popular ssh client[2].
Software has been developed to detect memory related issues. In this paper we want to present *Valgrind*[3] and its tool *Memcheck*, that are able to detect the double free error. We will illustrate how Valgrind identifies double frees and how a user can utilize Valgrind to find and fix double frees.

```c
void simple_double_free(){
    int *p = (int*) malloc(sizeof(int));

    free(p);
    free(p);
}
```

```c
void complex_double_free(int i){
    int *p1 = (int*) malloc(sizeof(int));
    int *p2 = p1;

    if(i > 0)
        free(p1);

    /*
        a lot of code
    */

    free(p1);
    free(p2);
}
```

Figure 1: examples of double free in c

1.1 Related work

Debugging, the process of finding origins of failures (also known as software *bugs*), has been studied quite well in literature. There is a lot of programming language specific literature that guides developers in avoiding errors in the first place. Dennis Ritchie and Brian Kernighan[4] give an overview of the most important features on *C*, including pointers and dynamic memory. Brian W. Kernighan and Rob Pike[5] presented an overview of developing and debugging software and proposed principles and strategies for different programming languages. Andreas Zeller[6] gives a guide on debugging and proposes the *TRAFFIC*-principle, a systematic approach for correcting failures. One strategy for finding bugs is the use of debugging software. One family of debugging software are debuggers such as *GDB*, the GNU Project Debugger[7]. They allow devel-

[1] Awakened. *How a double-free bug in WhatsApp turns to RCE*. Last accessed 14 May 2023. 2019. URL: `https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/`.
[2] OpenSSH. *OpenSSH security log*. Last accessed 14 May 2023. URL: `https://www.openssh.com/security.html`.
[3] *Valgrind homepage*. URL: `https://valgrind.org/`.
[4] D.M. Ritchie and B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.
[5] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison–Wesley, 1999.
[6] Zeller Andreas. *Why Programs Fail : A Guide to Systematic Debugging*. Elsevier, 2006.
[7] R. Stallman, R.H. Pesch, and S. Shebs. *Debugging with GDB*. Free Software Foundation, 2011.

opers to stop a program at specific points and lets the developer examine the state and the trace of the program. Another type of debugging software are tools that analyze the code statically or dynamically at runtime. Valgrind is an instrumentation framework for building dynamic analysis tools[8]. Memcheck is a tool for Valgrind for finding memory related bugs and has been described and evaluated with the focus on undefined value errors[9]. More recently the capabilities of Valgrind in parallel environments have been studied[10].

2 Methods

We evaluate Memcheck and its capability to detect double free errors by running it against faulty code such as in figure 1.
When Memcheck detects an error, it generates a detailed report that includes information about the type of error, the location in the code where it occurred, and the call stack leading up to the error.
We then analyze and discuss the created report for the faulty code.
Our investigation of Valgrind is supported by researching literature such as the provided documentation and manual by Valgrind as well as related scientific papers. Additionally since Valgrind is open source, we can study and analyze the source code.

```
1  valgrind ./double_free
```

Figure 2: Invoking Valgrind on an executable

3 Evaluation

3.1 Testbed

We will be using Memcheck, a tool under Valgrind, and running it on Harris. The code will be written in C.

3.2 Results

3.2.1 Using Valgrind to detect double free errors

Running the faulty example program on our testbed fails with the following output:

[8]Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: https://doi.org/10.1145/1273442.1250746.

[9]Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision". In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit.

[10]Daniel Robson and Peter Strazdins. "Parallelisation of the Valgrind Dynamic Binary Instrumentation Framework". In: *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. 2008, pp. 113–121. DOI: 10.1109/ISPA.2008.94.

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

The output includes the reason of the failure, a double free, however it does not give any context on where it happened. This makes it difficult to debug. Also keep in mind that the behavior of double free is undefined. On other machines or for more complex code it possible that no error occurs or that the failure appears at a different point in the program.
When invoking Valgrind with Memcheck on a program (also see figure 2), it outputs a commentary of the run. For our program the output is:

```
==644167== Memcheck, a memory error detector
==644167== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==644167== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==644167== Command: ./double_free
==644167==
==644167== Invalid free() / delete / delete[] / realloc()
==644167==    at 0x484B27F: free (in /.../valgrind/vgpreload_memcheck-amd64-linux.so)
==644167==    by 0x10925E: main (double_free.c:45)
==644167==  Address 0x4a96040 is 0 bytes inside a block of size 8 free'd
==644167==    at 0x484B27F: free (in /.../valgrind/vgpreload_memcheck-amd64-linux.so)
==644167==    by 0x109252: main (double_free.c:44)
==644167==  Block was alloc'd at
==644167==    at 0x4848899: malloc (in /.../valgrind/vgpreload_memcheck-amd64-linux.so)
==644167==    by 0x109238: main (double_free.c:40)
==644167==
==644167==
==644167== HEAP SUMMARY:
==644167==     in use at exit: 0 bytes in 0 blocks
==644167==   total heap usage: 1 allocs, 2 frees, 4 bytes allocated
==644167==
==644167== All heap blocks were freed -- no leaks are possible
==644167==
==644167== For lists of detected and suppressed errors, rerun with: -s
==644167== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

At the start of each line is the ID of the process that was responsible for outputting the line. This makes the difference between Valgrind and program output clear. Since our program does not print anything, the ID is identical in each line.
The comment begins with general information like a title and version (lines one to four). Then a list of errors (if any were found) starts. In our case you can see in line six that there is an invalid free, delete or realloc in our program. The next line confirms that it is free and gives the function and file name where the error occurred. If the program was compiled with debugging symbols (-g option in GCC), there will also be the actual line number of the source code. In a similar manner it includes where the memory was freed before (see line 10-13) and where it the memory was allocated (see line 14-17). This information simplifies tracking down the chain of infection to find the true defect.
Lines 20-24 are a summary of the heap usage, indicating that there were no bytes in use at the exit and a total of 1 allocation and 2 frees. It also includes information for leaks, another common misuse with dynamic memory.

The commentary can be used to locate and correct the error by looking at the specified lines. After correcting the code by removing the additional free(p), we can validate the program with Valgrind again:

```
1  void simple_double_free(){
2     int *p = (int*) malloc(sizeof(int));
3
4    free(p);
5  }
```

Figure 3: Correction of code for simple_double_free()

```
==644381== Memcheck, a memory error detector
==644381== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==644381== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==644381== Command: ./main
==644381==
==644381==
==644381== HEAP SUMMARY:
==644381==     in use at exit: 0 bytes in 0 blocks
==644381==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==644381==
```

```
11 ==644381== All heap blocks were freed -- no leaks are possible
12 ==644381==
13 ==644381== For lists of detected and suppressed errors, rerun with: -s
14 ==644381== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Line 14 confirms that there are no more errors.

We showed how to use Valgrind and and explained how the created comment can be used to correct double frees. An advantage of Valgrind is, that the behavior is independent of the users environment. In the next section we will analyze how Valgrind is able to detect double free errors.

3.2.2 Detecting double free errors

Valgrind enables its tools to intercept function calls and lets them add additional instructions. Memcheck replaces among others the function definitions for `malloc`, `calloc`, `realloc` and `free` and adds additional functionality, used for ensuring their correct use. Internally Memcheck keeps two data structures for tracking all malloc'd and free'd blocks.
When handling a `malloc` or `calloc`, additionally the returned address and some additional information such as the length of the memory block is saved and put into the malloc'd blocks data structure.
When a `free` is used, Memcheck tests if the address has been allocated before by searching through the malloc'd blocks. If it finds an corresponding entry, it removes the entry from the malloc's blocks and puts into free'd blocks data structure. If it cannot find the memory block, the freeing of the block must be invalid and an error is reported. This ensures that each memory block is freed exactly once.
Similar is the behavior for calls to `realloc`. If the memory is not present in the malloc'd blocks, again an error is created. Else the malloc'd blocks entry is updated.
When reporting an illegal `free` or `realloc`, Memcheck uses the stored data sources, to trace back the allocation history. To be performant a hashing data structure is used for the malloc'd and free'd blocks. Since the size of the free'd data structure is limited, Memcheck also handles overfilling by evicting old entries.

This way Valgrind is able to detect double free errors reliable.

3.2.3 Using Valgrind to detect no free errors

Next, we explored further how Valgrind detects no freeing errors. A "no freeing" error indicates a memory leak in the program. A memory leak occurs when dynamically allocated memory is not properly deallocated or freed, resulting in unused memory that cannot be accessed or released for reuse. A simple function of no free can be written (see figure5) and running the program will not return any errors, which is risky. Thus, using Valgrind with Memcheck to run the program will provide more information.
The function of no free:

```c
void no_freeing(void)
{
    int* arr = malloc(10 * sizeof(int));
}
```

Figure 4: Simple example of no free function

```
==1197138== Memcheck, a memory error detector
==1197138== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1197138== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1197138== Command: ./main
==1197138==
==1197138==
==1197138== HEAP SUMMARY:
==1197138==     in use at exit: 40 bytes in 1 blocks
==1197138==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==1197138==
==1197138== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1197138==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-lin
==1197138==    by 0x10919E: no_freeing (in /home/a12242501/exercises/valgrind/main)
==1197138==    by 0x109245: main (in /home/a12242501/exercises/valgrind/main)
==1197138==
==1197138== LEAK SUMMARY:
==1197138==    definitely lost: 40 bytes in 1 blocks
==1197138==    indirectly lost: 0 bytes in 0 blocks
==1197138==      possibly lost: 0 bytes in 0 blocks
==1197138==    still reachable: 0 bytes in 0 blocks
==1197138==         suppressed: 0 bytes in 0 blocks
==1197138==
==1197138== For lists of detected and suppressed errors, rerun with: -s
==1197138== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Line 11 indicates that there is a memory leak in the program. The memory allocated with malloc (or a similar function) is no longer accessible by the program, so there is no way to free it. The program has lost track of this memory, hence the "definitely lost" classification. In addition. line 12 specifies the location where the memory was located where the allocation happened inside the malloc function. Line 13 further identifies the specific location that points out the function no_freeing. Lastly, Valgrind summarizes the total number of errors. In this case, there is one error, which is the memory leak.

By glancing at the designated lines in the commentary, the error can be found and fixed. We can revalidate the program with Valgrind after fixing the code by adding free(arr):

```c
void no_freeing(void)
{
    int* arr = malloc(10 * sizeof(int));
    free(arr);
}
```

Figure 5: Correction of code for no_freeing()

```
==1197245== Memcheck, a memory error detector
==1197245== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1197245== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1197245== Command: ./main
==1197245==
==1197245==
==1197245== HEAP SUMMARY:
==1197245==     in use at exit: 0 bytes in 0 blocks
==1197245==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==1197245==
==1197245== All heap blocks were freed -- no leaks are possible
==1197245==
==1197245== For lists of detected and suppressed errors, rerun with: -s
==1197245== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Line 14 confirms that there are no more errors.

3.2.4 Detecting reading outside of the malloc'd blocks

Following on, we explored further how Valgrind detects reading outside of the malloc'd blocks. A simple function can be written (see figure6) and running the program will not return any errors, which is dangerous. It can lead to data corruption, accessing memory beyond what has been allocated. This includes overwriting or corrupting data that belongs to other variables or data structures. Thus, using Valgrind with Memcheck to run the program will provide more information.

```
1  int read_end_malloc_block(void)
2  {
3      int* arr = (int*)malloc(2 * sizeof(int));
4      for (int i = 0; i < 2; i++) {
5          arr[i] = i;
6      }
7      printf("%d ", arr[5]);
8      free(arr);
9
10     return 0;
11 }
```

Figure 6: Simple example of reading outside of the malloc'd blocks

```
1  ==1197739== Memcheck, a memory error detector
2  ==1197739== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3  ==1197739== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
4  ==1197739== Command: ./main
5  ==1197739==
6  ==1197739== Invalid read of size 4
7  ==1197739==    at 0x109293: read_end_malloc_block (in /home/a12242501/exercises/valgrind/m
8  ==1197739==    by 0x1092CA: main (in /home/a12242501/exercises/valgrind/main)
9  ==1197739==  Address 0x4a96054 is 12 bytes after a block of size 8 alloc'd
10 ==1197739==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-lin
11 ==1197739==    by 0x10925A: read_end_malloc_block (in /home/a12242501/exercises/valgrind/m
12 ==1197739==    by 0x1092CA: main (in /home/a12242501/exercises/valgrind/main)
13 ==1197739==
14 0 ==1197739==
15 ==1197739== HEAP SUMMARY:
16 ==1197739==     in use at exit: 0 bytes in 0 blocks
17 ==1197739==   total heap usage: 2 allocs, 2 frees, 1,032 bytes allocated
18 ==1197739==
19 ==1197739== All heap blocks were freed -- no leaks are possible
20 ==1197739==
21 ==1197739== For lists of detected and suppressed errors, rerun with: -s
22 ==1197739== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind has provided a detailed summary of errors found in the function. Line 6 indicates that the program is trying to read an element beyond the array size of 4. Line 7 specifies the location where the error is found in, read_end_malloc_block function. Line 9 indicates that the invalid read occurred 12 bytes after a block of size 8 was allocated.

After correcting the code by changing arr[5] to arr[1], we can validate the program with Valgrind again:

```c
int read_end_malloc_block(void)
{
    int* arr = (int*)malloc(2 * sizeof(int));
    for (int i = 0; i < 2; i++) {
        arr[i] = i;
    }
    printf("%d ", arr[1]);
    free(arr);

    return 0;
}
```

Figure 7: Correction of code for read_end_malloc_()

```
==1198811== Memcheck, a memory error detector
==1198811== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1198811== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1198811== Command: ./main
==1198811==
1 ==1198811==
==1198811== HEAP SUMMARY:
==1198811==     in use at exit: 0 bytes in 0 blocks
==1198811==   total heap usage: 2 allocs, 2 frees, 1,032 bytes allocated
==1198811==
==1198811== All heap blocks were freed -- no leaks are possible
==1198811==
==1198811== For lists of detected and suppressed errors, rerun with: -s
==1198811== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Line 14 confirms that there are no more errors.

3.2.5 Accessing outside the array length

Lastly, we found out that Valgrind is unable to detect any errors when an array is not assigned by malloc (i.e. p[4] = 1, 2, 3, 4;). A simple function written (see figure8 of different initialization of an array will not return any errors when reading outside the bounds of an array.

```c
void access_out_of_bounds(){
    int p[4] = {1,2,3,4};

    printf("%d\n", p[10]);
}
```

Figure 8: Simple example of reading outside of the malloc'd blocks

```
==1026052== Memcheck, a memory error detector
==1026052== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1026052== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1026052== Command: ./main
==1026052==
1086295
==1026052==
==1026052== HEAP SUMMARY:
==1026052==     in use at exit: 0 bytes in 0 blocks
==1026052==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==1026052==
==1026052== All heap blocks were freed -- no leaks are possible
==1026052==
==1026052== For lists of detected and suppressed errors, rerun with: -s
==1026052== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind indicates that there are no memory errors or leaks detected by. The program was executed successfully without any issues. In addition, an output of 1086295 is displayed even though the size of the array is 4.

4 Discussion

In conclusion, this study aimed to evaluate the effectiveness of Valgrind and its Memcheck tool in detecting errors in software development. The results obtained indicate that Valgrind, with the assistance of Memcheck, is a powerful tool for identifying and diagnosing memory-related errors.

The evaluation demonstrated that Valgrind successfully detected a significant number of errors such as double free, no freeing and reading outside of malloc'd blocks in the tested code samples. By providing detailed reports and stack traces, Memcheck facilitated the identification of the specific code locations where the errors occurred, enabling developers to address them effectively. This finding highlights the importance of using dynamic analysis tools like Valgrind in the software development process to enhance code quality and reliability.

However, it is worth noting that Valgrind and Memcheck may have limitations in detecting array-related errors, such as out-of-bounds accesses when the memory for the array is allocated statically on the stack. Therefore, while Valgrind and Memcheck excel in detecting certain memory errors, developers should take note of its limitations.

Future research could focus on expanding the evaluation to include a wider range of codebases and programming languages. Additionally, incorporating other dynamic analysis tools or combining Valgrind with static analysis techniques may offer a more holistic approach to memory error detection.