**Memory Related Issues**

We might have faced different types of memory related issues in heap and stack segments like

* Memory leak
* Memory Overrunning or Underrunning
* Stack Overflow
* Heap overflow etc...

A **memory debugger** is a programming tool for finding memory leaks and buffer overflows. These are due to bugs related to the allocation and deallocation of dynamic memory.We have different types of memory debugging tools **VALGRIND** is one of them.

**Different Memory Debugging Tools:**

**AddressSanitizer** (or ASan) is an open source programming tool by google that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer(use-after-free). AddressSanitizer is based on compiler instrumentation and directly-mapped shadow memory.

**Bounds Checker** is a memory checking and API call validation tool used for C++ software development with Microsoft Visual C++.BoundsChecker may be run in two distinct modes: ActiveCheck, which will work against any application as is, or FinalCheck, which makes use of instrumentation added to the application when it is built.

**Electric Fence** (or eFence) is a memory debugger written by Bruce Perens. It consists of a library which programmers can link into their code to override the C standard library memory [management](https://en.wikipedia.org/wiki/Memory_management) functions. eFence triggers a program crash when the memory error occurs, so a [debugger](https://en.wikipedia.org/wiki/Debugger) can be used to inspect the code that caused the error.

**Valgrind** is a programming tool for memory debugging, memory leak detection, and profiling. Valgrind was originally designed to be a free memory debugging tool for Linux on x86, but has since evolved to become a generic framework for creating dynamic analysis tools such as checkers and profilers. It also contain number of additional utilities for performance profiling, finding synchronization errors in multi-threading programs and analysis of memory consumption.

**Valgrind's distribution contain following modules:**

**Memcheck**

Main module, that provide memory leak detection. This module also could be used for finding other errors of work with memory — read or write behind memory blocks boundaries, etc.

**Cachegrind**

Analyze execution of code, collecting data about processor cache misses and code branching (when processor has wrong prediction about jumping). This statistic collecting for all program, separate functions and lines of code

**Callgrind**

Analyze functions calls, using almost same methods, as cachegrind module. This allow to build tree of functions calls, and analyze performance

**Massif**

Allow to analyze memory consumption in different parts of program

**Helgrind**

Analyze executed code for presence of different synchronization errors in programs, that use POSIX Threads.

There are also some number of modules, but they are experimental. Users also can create their own modules, that could perform analysis of executed code.

**NOTE :** By default Valgrind run memcheck module, but user can select another module by specifying it with --tool option passing it the name of module, for example:

**valgrind [valgrind-options] your-prog [your-prog-options]**

**PROGRAMS:**

### **Use of uninitialized pointer and variable**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void)  {  ch \*ptr;  char ch = \*ptr;  printf("\n [%c]\n",ch);  return 0;  } |

gcc -g <file name> -o <executable file name>

valgrind ./<executable file name>

**NOTE:**

* If you want to know how many times each error occurred, run with the -v option.
* Use --track-origins=yes to see where uninitialised values come from.

### **RESULT:**

|  |
| --- |
| ==5308== **Use of uninitialised value of size 8**  ==5308== at 0x400539: main (in /home/kumbhshr/Memory/uninitialized)  ==5308==  ==5308== **Invalid read of size 1**  ==5308== at 0x400539: main (in /home/kumbhshr/Memory/uninitialized)  ==5308== Address 0x0 is not stack'd, malloc'd or (recently) free'd |

### **Use of uninitialized pointer and initialized variable**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void)  {  char \*ptr,ch = 's';  \*p = ch;  printf("\n [%c]\n",ch);  return 0;  } |

### **RESULT:**

|  |
| --- |
| ==5633== **Use of uninitialised value of size 8**  ==5633== at 0x400541: main (in /home/kumbhshr/Memory/uninitialized)  ==5633==  ==5633== **Invalid write of size 1**  ==5633== at 0x400541: main (in /home/kumbhshr/Memory/uninitialized)  ==5633== Address 0x0 is not stack'd, malloc'd or (recently) free'd |

### **Use of initialized pointer and variable**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void)  {  char \*ptr = (char\*)malloc(2);  char ch = 's';  \*p = ch;  printf("\n [%c]\n",ch);  free(ptr);  return 0;  } |

### **RESULT: No Error**

### **Memory Leak:**

### Memory is allocated but not released causing an application to consume memory reducing the available memory for other applications and eventually causing the system to page virtual memory to the hard drive slowing the application or crashing the application when than the computer memory resource limits are reached. The system may stop working as these limits are approached.

### **Example**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void)  {  char \*ptr = malloc(1);  char ch;  \*ptr = 'a';  ch = \*ptr;  printf("\n [%c]\n",ch);  return 0;  } |

### **Note:**

* To get more details of leaked memory add  **--leak-check=full** as valgrind option.

### **RESULT:**

|  |
| --- |
| ==5796== **HEAP SUMMARY:**  ==5796== in use at exit: 1 bytes in 1 blocks  ==5796== total heap usage: 1 allocs, 0 frees, 1 bytes allocated  ==5796==  ==5796== **LEAK SUMMARY:**  ==5796== definitely lost: 1 bytes in 1 blocks  ==5796== indirectly lost: 0 bytes in 0 blocks  ==5796== possibly lost: 0 bytes in 0 blocks  ==5796== still reachable: 0 bytes in 0 blocks  ==5796== suppressed: 0 bytes in 0 blocks |

### **Doubly Free Memory:**

### In the above peice of code, we have freed the memory pointed by ‘p’ twice. Now, lets run the tool memcheck :

### **Example**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void)  {  char \*ptr = (char\*)malloc(1);  \*ptr = 'a';  char ch = \*ptr;  printf("\n [%c]\n",ch);  free(ptr);  free(ptr);  return 0;  } |

### **RESULT:**

|  |
| --- |
| ==5919== **Invalid free() / delete / delete[] / realloc()**  ==5919== at 0x4C2BDEC: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==5919== by 0x400610: main (in /home/kumbhshr/Memory/doublefree)  ==5919== Address 0x5200040 is 0 bytes inside a block of size 1 free'd  ==5919== at 0x4C2BDEC: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==5919== by 0x400604: main (in /home/kumbhshr/Memory/doublefree)  ==5919==  ==5919==  ==5919== **HEAP SUMMARY:**  ==5919== in use at exit: 0 bytes in 0 blocks  ==5919== total heap usage: 1 allocs, 2 frees, 1 bytes allocated |

### **Dangling Pointer:**

### Dangling pointers arise during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

### **Example**

|  |
| --- |
| #include<stdio.h>  #include <stdlib.h>    int main(void)  {  int \*ptr = malloc(10 \* sizeof(int));  int num=10;  printf("%p\n",ptr);  \*ptr = num;  printf("%d\n",\*ptr);  free(ptr);  \*ptr = num;  printf("%d\n",\*ptr);  return 0;  } |

### **RESULT:**

|  |
| --- |
| 0x5200040  10  ==7998== **Invalid write of size 4**  ==7998== at 0x400623: main (in /home/kumbhshr/Memory/memory)  ==7998== Address 0x5200040 is 0 bytes inside a block of size 40 free'd  ==7998== at 0x4C2BDEC: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==7998== by 0x40061B: main (in /home/kumbhshr/Memory/memory)  ==7998==  ==7998== **Invalid read of size 4**  ==7998== at 0x400629: main (in /home/kumbhshr/Memory/memory)  ==7998== Address 0x5200040 is 0 bytes inside a block of size 40 free'd  ==7998== at 0x4C2BDEC: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==7998== by 0x40061B: main (in /home/kumbhshr/Memory/memory)  ==7998==  10  ==7998== |

### **Stack Overflow:**

### A stack overflow is an undesirable condition in which a particular computer program tries to use more memory space than the call stack has available. In programming, the call stack is a buffer that stores requests that need to be handled.

### **Example**

|  |
| --- |
| #include<stdio.h>  void function1();  void function();    int main()  {  function();  }    void function1()  {  function();  }    void function()  {  function1();  } |

### **RESULT:**

|  |
| --- |
| **Stack overflow in thread 1: can't grow stack to 0xffe801ff8**  ==8336==  ==8336== Process terminating with default action of signal 11 (SIGSEGV)  ==8336== Access not within mapped region at address 0xFFE801FF8  ==8336== at 0x400506: ping (in /home/kumbhshr/Memory/stackoverflow)  ==8336== If you believe this happened as a result of a stack  ==8336== overflow in your program's main thread (unlikely but  ==8336== possible), you can try to increase the size of the  ==8336== main thread stack using the --main-stacksize= flag.  ==8336== The main thread stack size used in this run was 8388608.  ==8336== **Stack overflow in thread 1: can't grow stack to 0xffe801ff0** |

### **Heap Overflow:**

### A heap overflow is a type of buffer overflow that occurs in the heap data area. Heap overflows are exploitable in a different manner to that of stack-based overflows. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data.

### **Example:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  void heapoverrun(void)  {  int\* ptr = malloc(10 \* sizeof(int));  ptr[10] = 0;  free(ptr);  }  int main(void)  {  int index;  for(index=0;index<5;index++)  heapoverrun();  return 0;  } |

### **RESULT:**

|  |
| --- |
| **Invalid write of size 4**  ==3128== at 0x40059B: heapoverrun (memory.c:5)  ==3128== by 0x4005C4: main (memory.c:12)  ==3128== Address 0x5200068 is 0 bytes after a block of size 40 alloc'd  ==3128== at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==3128== by 0x40058E: heapoverrun (memory.c:4)  ==3128== by 0x4005C4: main (memory.c:12) |

### **Overlapping source and destination blocks**

### **Example:**

|  |
| --- |
| #include<string.h>    int main()  {  char buf[1000];  memcpy(buf,buf+10,200);  } |

### **RESULT:**

|  |
| --- |
| ==5825== **Source and destination overlap in memcpy(0xffefff940, 0xffefff94a, 200)**  ==5825== at 0x4C2F71C: memcpy@@GLIBC\_2.14 (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==5825== by 0x4005F9: main (in /home/kumbhshr/valgrindex/a.out)  ==5825==  ==5825==  ==5825== HEAP SUMMARY:  ==5825== in use at exit: 0 bytes in 0 blocks  ==5825== total heap usage: 0 allocs, 0 frees, 0 bytes allocated  ==5825==  ==5825== All heap blocks were freed -- no leaks are possible  ==5825==  ==5825== For counts of detected and suppressed errors, rerun with: -v  ==5825== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0) |