Valgrind Tutorial

Memory Errors and Valgrind

While dynamic memory is a powerful technique that gives a programmer the ability to create complex data structures, it can be very prone to memory errors. Valgrind is a useful tool to help track down memory leaks and other memory related errors.
There are several major types of memory related errors:

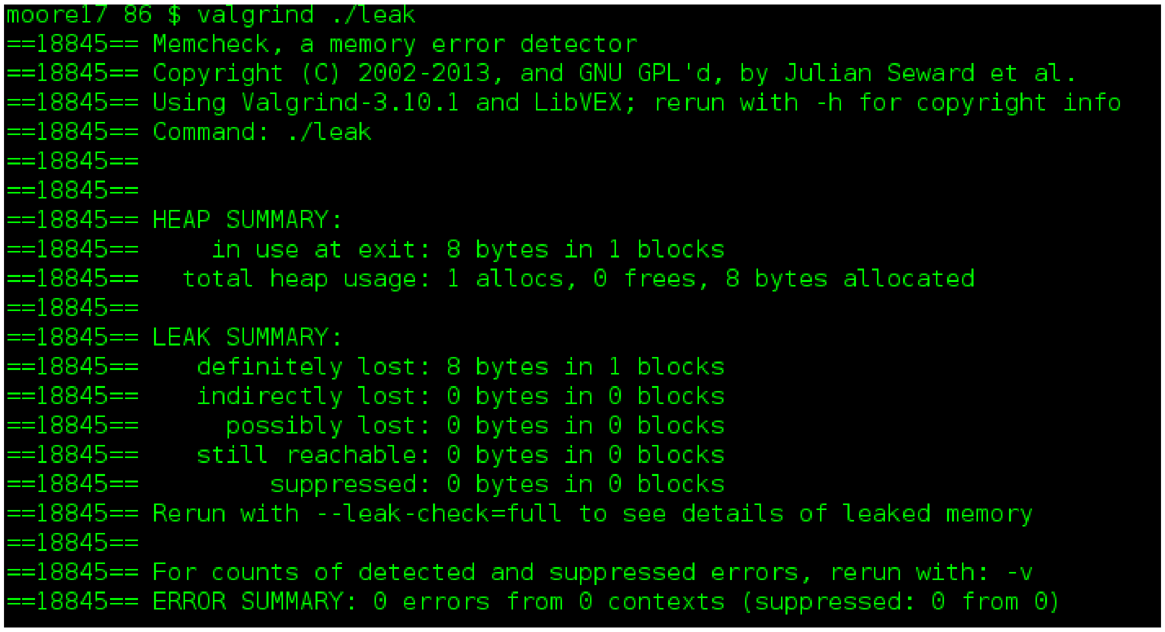
- Memory Leaks
- Invalid Read
- Invalid Write
- Uninitialized Read
- Use-After-Free (Dangling Pointer)

Memory Leaks

A memory leak occurs when a program continually calls malloc(), but forgets to call free() when it is done using the memory. Consider the following program that leaks memory:

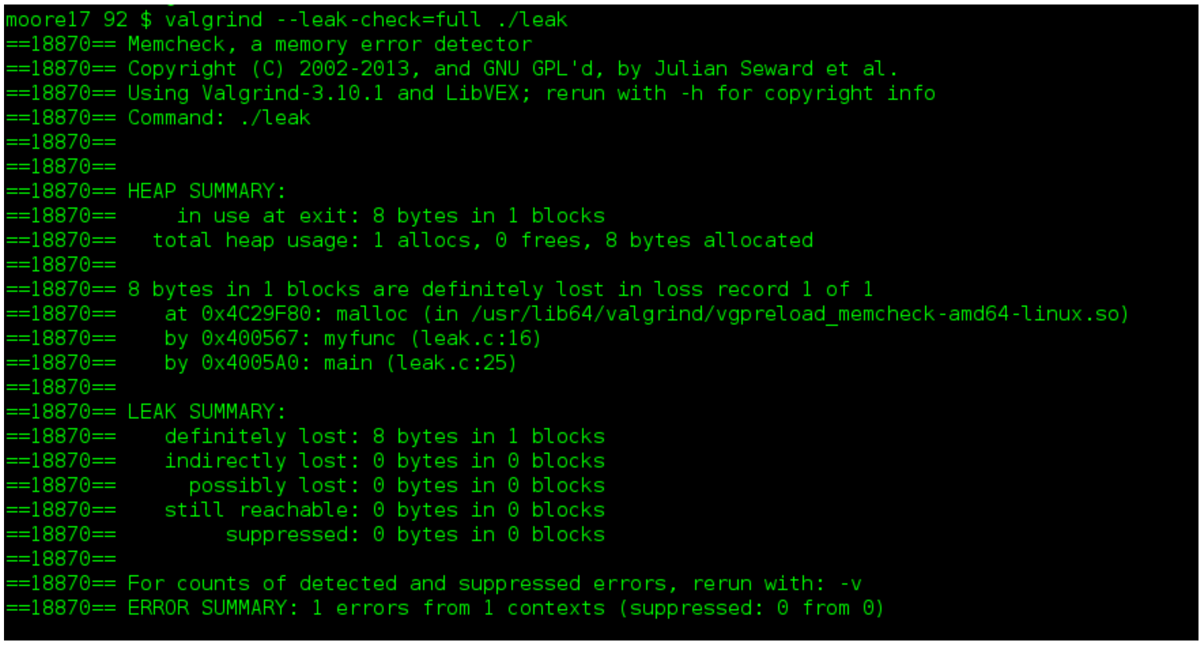
```c
#include <stdlib.h>

/**
 * Example struct
 */
struct mystruct {
    int somedata;
    int moredata;
};

/**
 * This function will leak memory
 */
int myfunc() {
    /* Allocate a struct on the heap for some purpose */
    struct mystruct* data = (struct mystruct*)malloc(sizeof(struct mystruct));
    data->somedata = 4;
    data->moredata = 5;

    /* Ooops, function exiting without freeing and we no longer have a copy of the pointer */
    return 0;
}

int main(int argc, char** argv) {
    myfunc();
    return 0;
}
```

If we compile the code into a program ‘**leak**’ and run with valgrind (‘**valgrind ./leak**’), we get the following output.



```
moore17 86 $ valgrind ./leak
==18845== Memcheck, a memory error detector
==18845== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==18845== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==18845== Command: ./leak
==18845==
==18845==
==18845== HEAP SUMMARY:
==18845==     in use at exit: 8 bytes in 1 blocks
==18845==   total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==18845==
==18845== LEAK SUMMARY:
==18845==    definitely lost: 8 bytes in 1 blocks
==18845==    indirectly lost: 0 bytes in 0 blocks
==18845==      possibly lost: 0 bytes in 0 blocks
==18845==    still reachable: 0 bytes in 0 blocks
==18845==         suppressed: 0 bytes in 0 blocks
==18845== Rerun with --leak-check=full to see details of leaked memory
==18845==
==18845== For counts of detected and suppressed errors, rerun with: -v
==18845== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

We can re-run valgrind with the '**--leak-check=full**' option to see more detailed information:



```
moore17 92 $ valgrind --leak-check=full ./leak
==18870== Memcheck, a memory error detector
==18870== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==18870== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==18870== Command: ./leak
==18870==
==18870==
==18870== HEAP SUMMARY:
==18870==     in use at exit: 8 bytes in 1 blocks
==18870==   total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==18870==
==18870== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==18870==    at 0x4C29F80: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==18870==    by 0x400567: myfunc (leak.c:16)
==18870==    by 0x4005A0: main (leak.c:25)
==18870==
==18870== LEAK SUMMARY:
==18870==    definitely lost: 8 bytes in 1 blocks
==18870==    indirectly lost: 0 bytes in 0 blocks
==18870==      possibly lost: 0 bytes in 0 blocks
==18870==    still reachable: 0 bytes in 0 blocks
==18870==         suppressed: 0 bytes in 0 blocks
==18870==
==18870== For counts of detected and suppressed errors, rerun with: -v
==18870== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

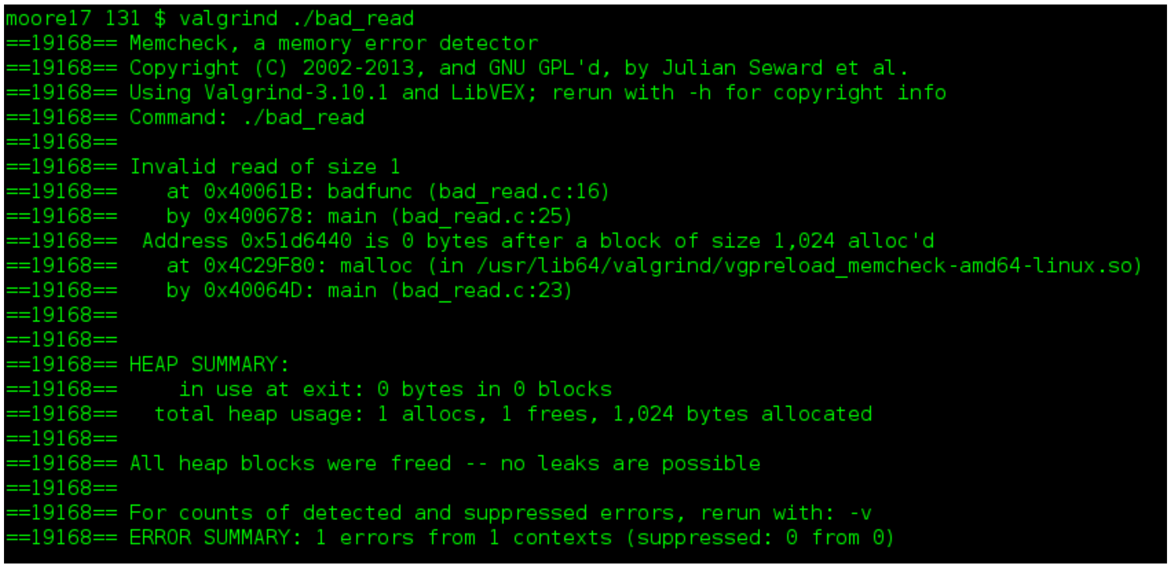
Here we can see that a block of memory allocated by a call to malloc() at line 16 in leak.c was never freed before the program terminated and thus is a memory leak.

Invalid Read

An invalid read is when a program read memory outside of valid memory.
Invalid reads commonly occur when working with arrays when an invalid index is used to access the array. Consider the following program that performs an invalid read past the end of an array:

```c
 1. #include <stdlib.h>
 2. #include <string.h>
 3.
 4.  /**
 5.   * This function performs an invalid read operation
 6.   */
 7.  int badfunc(char *arr, int len) {
 8.      // Always remember to check parameters
 9.      if (arr == NULL || len < 0) {
10.          return -1;
11.      }
12.      int sum;
13.      int i;
14.      // The condition should be i < len
15.      for (i = 0; i <= len; i++) {
16.          sum += arr[i];
17.      }
18.      return sum;
19.  }
20.
21.  int main(int argc, char** argv) {
22.      // Allocate a buffer for use
23.      char *buffer = (char*)malloc(1024);
24.      memset(buffer, 1, 1024);
25.      badfunc(buffer, 1024);
26.      // Free the buffer now that we are done with it
27.      free(buffer);
28.      return 0;
29.  }
```

Compiling this code into a program named ‘**bad_read**’ and running it with valgrind via ‘**valgrind ./bad_read**’ gives the following output:



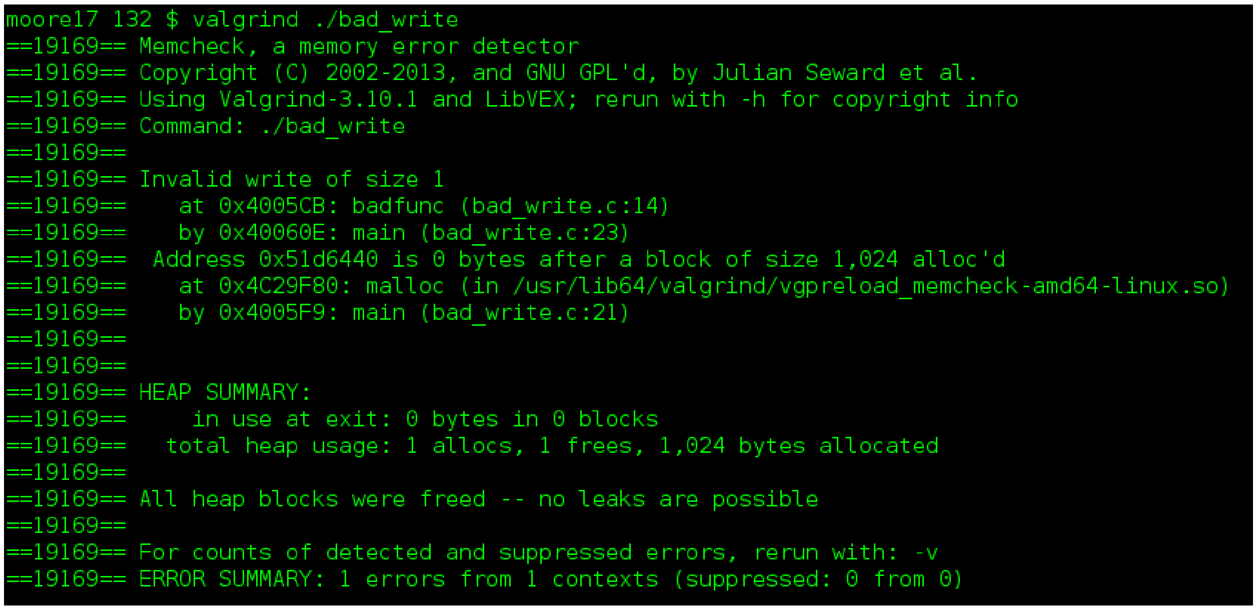
```
moore17 131 $ valgrind ./bad_read
==19168== Memcheck, a memory error detector
==19168== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==19168== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==19168== Command: ./bad_read
==19168==
==19168== Invalid read of size 1
==19168==    at 0x40061B: badfunc (bad_read.c:16)
==19168==    by 0x400678: main (bad_read.c:25)
==19168==  Address 0x51d6440 is 0 bytes after a block of size 1,024 alloc'd
==19168==    at 0x4C29F80: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==19168==    by 0x40064D: main (bad_read.c:23)
==19168==
==19168==
==19168== HEAP SUMMARY:
==19168==     in use at exit: 0 bytes in 0 blocks
==19168==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==19168==
==19168== All heap blocks were freed -- no leaks are possible
==19168==
==19168== For counts of detected and suppressed errors, rerun with: -v
==19168== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Invalid Write

Invalid writes can be caused by many different errors, including but not limited to dereferencing a pointer that was already freed, or by incorrectly indexing past the end of an array. Consider the following program that writes to invalid memory, causing corruption:

```c
1.  #include <stdlib.h>
2.
3.   /**
4.     * This function performs an invalid write operation
5.     */
6.   int badfunc(char *arr, int len) {
7.       // Always remember to check parameters
8.       if (arr == NULL || len < 0) {
9.           return -1;
10.      }
11.      int i;
12.      // The condition should be i < len, so we write to an element one past the end of the array
13.      for (i = 0; i <= len; i++) {
14.          arr[i] = 1;
15.      }
16.      return 0;
17.  }
18.
19.  int main(int argc, char** argv) {
20.      // Allocate a buffer for use
21.      char *buffer = (char*)malloc(1024);
22.      // Pass the buffer to a function
23.      badfunc(buffer, 1024);
24.      // Free the buffer now that we are done with it
25.      free(buffer);
26.      return 0;
27.  }
```

While in this trivial case the program will continue to run correctly, any program that performs an invalid write operation that corrupts memory is liable to crash at a later point in time with no warning. Valgrind will give the following warning when running the above program with an invalid write:



```
moore17 132 $ valgrind ./bad_write
==19169== Memcheck, a memory error detector
==19169== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==19169== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==19169== Command: ./bad_write
==19169==
==19169== Invalid write of size 1
==19169==    at 0x4005CB: badfunc (bad_write.c:14)
==19169==    by 0x40060E: main (bad_write.c:23)
==19169==  Address 0x51d6440 is 0 bytes after a block of size 1,024 alloc'd
==19169==    at 0x4C29F80: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==19169==    by 0x4005F9: main (bad_write.c:21)
==19169==
==19169==
==19169== HEAP SUMMARY:
==19169==     in use at exit: 0 bytes in 0 blocks
==19169==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==19169==
==19169== All heap blocks were freed -- no leaks are possible
==19169==
==19169== For counts of detected and suppressed errors, rerun with: -v
==19169== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Use-After-Free (Dangling Pointer)

A dangling pointer is a form of invalid write or read. For a dangling pointer to occur, memory is allocated and used correctly before finally being free()ed as it should. The dangling pointer error is when an access is made to the memory that already been freed. It's referred to as a dangling pointer because you are using a pointer to memory that is no longer allocated (hence dangling) for use. The biggest problem with dangling pointers occurs when the memory that a dangling pointer points to is allocated again for a different purpose, then it is written to by another piece of code, and finally the dangling pointer is dereferenced. In the event you read a dereferenced dangling pointer, you'll read bad data, and in the case of a write to a dereferenced dangling pointer, you'll overwrite and corrupt the memory that was being used by the other piece of code. The following program is an example of a dangling pointer:

```c
#include <stdlib.h>

char *myfunc() {
    char* mydata = (char*)malloc(100);
    /* Make a copy of the pointer */
    char* cpy = mydata;
    if (mydata == NULL) return NULL;
    /* Do work with mydata */
    free(mydata);
    /* Do more work, forget that the memory pointed to by BOTH mydata AND cpy has been freed */
    return cpy;
}

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

    char* data = myfunc();
    if (data == NULL) return 1;
    /* This is a dangling pointer, the memory pointed to by data was freed inside myfunc */
    data[10] = '\0';
    return 0;
}
```

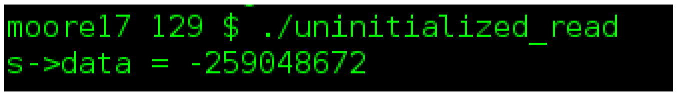
Uninitialized Memory

Another type of memory is when you read from uninitialized memory. When memory is allocated, either via malloc() or on the stack via a function call, the values in the area of memory are undefined, it can contain garbage values. Consider the following program that accesses uninitialized memory:

```c
#include <stdio.h>
 #include <stdlib.h>

 struct mystruct {
     int data;
     char moredata;
 };

 int badfunc(struct mystruct *s) {
     // Always check pointer parameters
     if (s == NULL) {
         return -1;
     }

     // s points to uninitialized memory, so this is an error
     printf("s->data = %d\n", s->data);
```

```c

    return 0;
}

int main(int argc, char** argv) {
    struct mystruct s;

    // Forgot to initialize s before passing the address
    badfunc(&s);

    return 0;
}
```

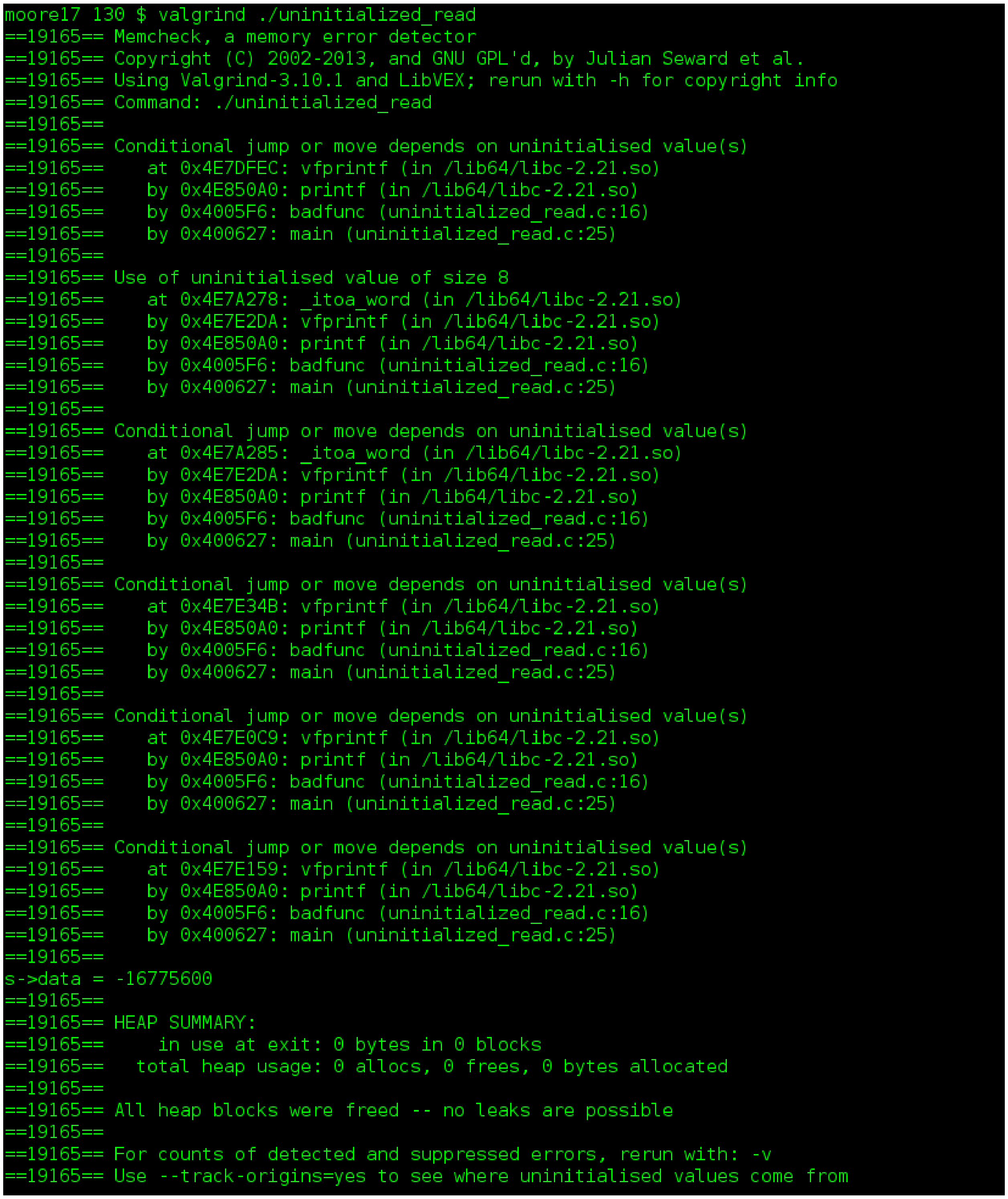
The following is an example of the output of a program compiled from the above code:



```
moore17 129 $ ./uninitialized_read
s->data = -259048672
```

Note how the value of **s->data** is random. This value can change each time the program is run based on what value happens to exist in memory.

Below is an example of Valgrind output indicating that an uninitialized variable is being used:



```
moore17 130 $ valgrind ./uninitialized_read
==19165== Memcheck, a memory error detector
==19165== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==19165== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==19165== Command: ./uninitialized_read
==19165==
==19165== Conditional jump or move depends on uninitialised value(s)
==19165==    at 0x4E7DFEC: vfprintf (in /lib64/libc-2.21.so)
==19165==    by 0x4E850A0: printf (in /lib64/libc-2.21.so)
==19165==    by 0x4005F6: badfunc (uninitialized_read.c:16)
==19165==    by 0x400627: main (uninitialized_read.c:25)
==19165==
==19165== Use of uninitialised value of size 8
==19165==    at 0x4E7A278: _itoa_word (in /lib64/libc-2.21.so)
==19165==    by 0x4E7E2DA: vfprintf (in /lib64/libc-2.21.so)
==19165==    by 0x4E850A0: printf (in /lib64/libc-2.21.so)
==19165==    by 0x4005F6: badfunc (uninitialized_read.c:16)
==19165==    by 0x400627: main (uninitialized_read.c:25)
==19165==
==19165== Conditional jump or move depends on uninitialised value(s)
==19165==    at 0x4E7A285: _itoa_word (in /lib64/libc-2.21.so)
==19165==    by 0x4E7E2DA: vfprintf (in /lib64/libc-2.21.so)
==19165==    by 0x4E850A0: printf (in /lib64/libc-2.21.so)
==19165==    by 0x4005F6: badfunc (uninitialized_read.c:16)
==19165==    by 0x400627: main (uninitialized_read.c:25)
==19165==
==19165== Conditional jump or move depends on uninitialised value(s)
==19165==    at 0x4E7E34B: vfprintf (in /lib64/libc-2.21.so)
==19165==    by 0x4E850A0: printf (in /lib64/libc-2.21.so)
==19165==    by 0x4005F6: badfunc (uninitialized_read.c:16)
==19165==    by 0x400627: main (uninitialized_read.c:25)
==19165==
==19165== Conditional jump or move depends on uninitialised value(s)
==19165==    at 0x4E7E0C9: vfprintf (in /lib64/libc-2.21.so)
==19165==    by 0x4E850A0: printf (in /lib64/libc-2.21.so)
==19165==    by 0x4005F6: badfunc (uninitialized_read.c:16)
==19165==    by 0x400627: main (uninitialized_read.c:25)
==19165==
==19165== Conditional jump or move depends on uninitialised value(s)
==19165==    at 0x4E7E159: vfprintf (in /lib64/libc-2.21.so)
==19165==    by 0x4E850A0: printf (in /lib64/libc-2.21.so)
==19165==    by 0x4005F6: badfunc (uninitialized_read.c:16)
==19165==    by 0x400627: main (uninitialized_read.c:25)
==19165==
s->data = -16775600
==19165==
==19165== HEAP SUMMARY:
==19165==     in use at exit: 0 bytes in 0 blocks
==19165==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==19165==
==19165== All heap blocks were freed -- no leaks are possible
==19165==
==19165== For counts of detected and suppressed errors, rerun with: -v
==19165== Use --track-origins=yes to see where uninitialised values come from
```

Note:

It is important to understand, Valgrind can only tell you if a program **execution** has a memory leak, not the program itself under every possible input. Students who do not thoroughly test all of their functions may lose points due to memory leaks simply because they never tested them enough.