Valgrind

Valgrind

- Valgrind is a memory mismanagement detector
- Shows memory leaks, deallocation errors, etc
- A wrapper around a collection of tools that do many other things (e.g., cache profiling)
- Here we focus on the default tool, memcheck

Memcheck

Can detect:

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions
- Some misuses of the POSIX pthreads API

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
        int i;
        int *a = malloc(sizeof(int) * 10);

        if (!a) return -1; /*malloc failed*/

        for (i = 0; i < 11; i++)
            a[i] = i;
        free(a);
        return 0;
}
```

- $ gcc -Wall -pedantic -g example1.c -o example
- $ valgrind ./example

==23779== Memcheck, a memory error detector ==23779== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al. ==23779== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info ==23779== Command: ./example ==23779== ==23779==

Invalid write of size 4 ==23779== at 0x400548: main (example1.c:9) ==23779== Address 0x4c30068 is 0 bytes after a block of size 40 alloc'd ==23779== at 0x4A05E46: malloc (vg_replace_malloc.c:195) ==23779== by 0x40051C: main (example1.c:6) ==23779== ==23779== ==23779==

HEAP SUMMARY: ==23779== in use at exit: 0 bytes in 0 blocks ==23779== total heap usage: 1 allocs, 1 frees, 40 bytes allocated ==23779== ==23779== All heap blocks were freed -- no leaks are possible ==23779== ==23779== For counts of detected and suppressed errors, rerun with: -v ==23779== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)

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

int main() {
        int i;
        int a[10];
        for (i = 0; i < 9; i++)
            a[i] = i;
        for (i = 0; i < 10; i++)
            printf("%d ", a[i]);

        return 0;
}
```

- ==24599== Conditional jump or move depends on uninitialised value(s)
- ==24599== at 0x33A8648196: vfprintf (in /lib64/libc-2.13.so) ==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
- ==24599== by 0x400567: main (example2.c:11)
- ==24599== ==24599== Use of uninitialised value of size 8 ==24599== at 0x33A864484B: _itoa_word (in /lib64/libc-2.13.so) ==24599== by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so) ==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so) ==24599== by 0x400567: main (example2.c:11)

- ==24599== ==24599== Conditional jump or move depends on uninitialised value(s) ==24599== at 0x33A8644855: _itoa_word (in /lib64/libc-2.13.so) ==24599== by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so) ==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so) ==24599== by 0x400567: main (example2.c:11) ==24599==

0 1 2 3 4 5 6 7 8 7

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

int main(){
        int i; int *a;
        for (i=0; i < 10; i++){
          a = malloc(sizeof(int) * 100);
         }
         free(a);
         return 0;
}
```

```
$ gcc -Wall -pedantic -g example3.c -o example3
$ valgrind --leak-check=full ./example3
==24810== HEAP SUMMARY:
==24810==     in use at exit: 3,600 bytes in 9 blocks
==24810==   total heap usage: 10 allocs, 1 frees, 4,000 bytes allocated
==24810==
==24810== 3,600 bytes in 9 blocks are definitely lost in loss record 1 of 1
==24810==    at 0x4A05E46: malloc (vg_replace_malloc.c:195)
==24810==    by 0x400525: main (example3.c:9)
==24810==
==24810== LEAK SUMMARY:
==24810==    definitely lost: 3,600 bytes in 9 blocks
==24810==    indirectly lost: 0 bytes in 0 blocks
==24810==      possibly lost: 0 bytes in 0 blocks
==24810==    still reachable: 0 bytes in 0 blocks
==24810==         suppressed: 0 bytes in 0 blocks
==24810==
==24810== For counts of detected and suppressed errors, rerun with: -v
==24810== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

- If you see leaks indicated as still reachable, this generally does not indicate a serious problem since the memory was probably still in use at the end of the program.
- Any leaks listed as "definitely lost" should be fixed (as should ones listed "indirectly lost" or "possibly lost"
- "indirectly lost" will happen if you do something like free the root node of a tree but not the rest of it
- "possibly lost" generally indicates the memory is actually lost
    - A function that allocates a buffer (perhaps to store a string) and returns it, but the caller never frees the memory after it is finished.
    - If a program like that runs for a long time, it will allocate a lot of memory that it does not need.

```
#include <stdio.h>
int main()
{
        char *p;
        // Allocation #1 of 19 bytes
        p = (char *) malloc(19);
        // Allocation #2 of 12 bytes
        p = (char *) malloc(12);
        free(p);
        // Allocation #3 of 16 bytes
        p = (char *) malloc(16);
        return 0;
}
```

```
gcc -o test -g test.c

valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./test
```

```
==9704== Memcheck, a memory error detector for x86-linux.
==9704== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==9704== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==9704== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==9704== For more details, rerun with: -v
==9704== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==9704== malloc/free: in use at exit: 35 bytes in 2 blocks.
==9704== malloc/free: 3 allocs, 1 frees, 47 bytes allocated.
==9704== For counts of detected errors, rerun with: -v
==9704== searching for pointers to 2 not-freed blocks.
==9704== checked 1420940 bytes.
```

```
==9704== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x80483BF: main (test.c:15)
==9704==
==9704==
==9704== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x8048391: main (test.c:8)
==9704==
==9704== LEAK SUMMARY:
==9704==    definitely lost: 35 bytes in 2 blocks.
==9704==    possibly lost:   0 bytes in 0 blocks.
==9704==    still reachable: 0 bytes in 0 blocks.
==9704==         suppressed: 0 bytes in 0 blocks.
```

- Allocation #1 (19 byte leak) is lost because p is pointed elsewhere before the memory from Allocation #1 is free'd.
- To help us track it down, Valgrind gives us a stack trace showing where the bytes were allocated. In the 19 byte leak entry, the bytes were allocate in test.c, line 8.

- Allocation #2 (12 byte leak) doesn't show up in the list because it is free'd.

- Allocation #3 shows up in the list even though there is still a reference to it (p) at program termination. This is still a memory leak! Again, Valgrind tells us where to look for the allocation (test.c line 15).

```
#include <stdlib.h>

 void f(void)
 {
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
 }                   // problem 2: memory leak -- x not freed

 int main(void)
 {
   f();
   return 0;
 }
```

What valgrind is not:

It does not do bounds checking on stack/static arrays (those not allocated with malloc)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int i;
        int x = 0;
        int a[10];
        for (i = 0; i < 11; i++)
                a[i] = i;
        printf("x is %d\n", x);
        return 0;
}
```

What valgrind is not:

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


int main(){
    char *str = malloc(10);
    gets(str);
    printf("%s\n",str);
    return 0;
}
```

- valgrind checks programs *dynamically* -- that is, it checks during actual program execution whether any leaks actually occurred for that execution
  - you run valgrind on a particular set of inputs that does not cause any bad memory accesses or memory to be leaked, valgrind will not report any errors, even though your program does contain bugs
- For a long string, this will overflow
- Strings of length less than 10 are fine
- Never use gets