Chapter 43
Memory Error Checking Tools

Revision 1.0, February 2026.

Objectives

- Learn about history and background of memory checking tools.
- Understand how these tools work.
- Learn how to use and interpret the results from the popular AddressSanitizer memory checking tool.

43.1 Introduction

The use of pointers and arrays in languages like C and C++ is error prone. And memory errors can be extremely hard to diagnose and find. As we discussed in Chapter 9, there are a variety of ways that a programmer easily can make mistakes that allow their code to be exploited. And these errors have been the root cause of serious security problems since the 1980's.

As we saw in Chapters 26 and 27, compiler writers and operating system designers have been busy producing improvements that should make memory errors less frequent and more detectable. With the experience of more than 40 years of pointer and memory errors, and with the recent compiler and operating system improvements, we would expect this type of error to be a thing of the past.

However, we can see from vulnerabilities found in real code that these memory errors (weaknesses) contribute to a worrisome number of recent vulnerabilities. The most recent Top 25 CWE data[1] shows that memory errors are still widely present in modern code. As long as we use C and C++, this type of error does not seem to be going away.

There is more than you can do, though, to reduce the likelihood that you will write code with such errors. You can get an additional layer of protection by using a dynamic memory checking tool such as AddressSanitizer[2]. This type of tool is also known as a memory debugger.

Some of the types of errors that can be detected by these tools include

- Buffer overflows and overruns for stack, global, and heap variables.
- Use of a pointer after the memory to which it points is freed.
- Double freeing of a pointer.

[1] `https://cwe.mitre.org/top25/`
[2] `https://github.com/google/sanitizers/wiki/AddressSanitizer`

- Memory leaks, where your program allocates memory but never frees it.

43.2 How Do These Tools Work?

Memory checking tools are dynamic, so they actually run your program, checking it for erroneous use of dynamic memory allocation, deallocation, pointer use, and array subscript use.

Memory checking tools work by augmenting both the allocated data and machine instructions that reference that data. Extra data fields are created around allocated memory to detect overflow, more detailed information about memory allocations is kept to detect errors in allocating and freeing memory, and extra instructions are inserted into the binary (machine language) output of the compiler. This extra code checks for a variety of errors such as buffer overflows on the stack, heap, or global memory, and memory allocation errors such as double `free` operations.

The extra code inserted into your program helps detect these insidious memory errors, but also makes it run slower. As a result, memory checking is only enabled during the coding, debugging, and testing stages of program development. Most production releases of code have such checking disabled.

43.3 Background on AddressSanitizer

AddressSanitizer, also known as Asan, is an open source project started by Google and first described in the published literature in 2012[3]. It has become enormously popular, being incorporated into well known and widely used compilers such as open source gcc and clang, and Microsoft MSVC.

Since AddressSanitizer is free, easy to use, and widely available, we will use it to present examples of tool use. The concepts shown here apply to other similar tools.

43.4 Building Your Program to Run AddressSanitizer

When you compile your program with the `clang` or `gcc` compilers, you can add options that will control the insertion of AddressSanitizer functionality into your code. For example, if you were using gcc to compile a C program in file `testprog.c`, you could run the following command to enable memory checking with AddressSanitizer:

```
gcc -g -O0 -o testprog -fsanitize=address testprog.c
```

[3] Konstantin Serebryany, Derek Bruening, Alexander Potapenko and Dmitry Vyukov, "Address Sanitizer: A Fast Address Sanity Checker", *2012 USENIX Annual Technical Conference,* Boston, Mass., June 2012.

Here is an explanation of each option and its purpose:

`-g`	AddressSanitizer will include file names and line numbers in its report if this debug flag is included.
`-O0`	Reduces the amount of code optimization done by the compiler. Using this option can make the output of AddressSanitizer more closely correspond to the code as you wrote it.
`-o testprog`	Specifies the name of the object file for the program being compiled.
`-fsanitize=address`	Enables AddressSanitizer functionality. This is the minimum option that you must use to enable memory address checking.

We make note of a couple of important points. First, AddressSanitizer works the same way for the `clang` compiler as it does for `gcc`, with the same options. Second, while AddressSanitizer has many options that can control detailed aspects of how it works, for most uses you do not need to understand these options. However, for large, complex systems, you might have to use some of these options. Programs that use multiple threads require special care. The full description of these many options can be found on the compiler's manual page[4].

43.5 Examples of AddressSanitizer Reports

We now work through a series of simple programs, looking at the error contained in the program and then the output generated by the AddressSanitizer to report that error. Each program will allocate a single array (often called a buffer) and then access the locations in the array in a loop, until it references past the end of the array.

43.5.1 Example: Program with a Stack Buffer Overflow

The first program, shown in Figure 1, allocates the array `stack` on the stack by declaring it as a local variable. The program is then compiled with AddressSanitizer enabled, as described above.

When the program is run, the instrumentation inserted by AddressSanitizer checks each array reference and detects that there was a buffer overflow on the program's stack. The report from this run is shown in Figure 2. The message on the first line (shown in red) reports the error.

[4] `https://man7.org/linux/man-pages/man1/gcc.1.html`

```
05: int main(int argc, char *argv[])
06: {
07:    int stack[10];
08:    int i;
09:
10:    for (i=0; i<1000; i++)
11:       stack[i] = 7;
12:    exit (0);
13: }
```

Figure 1: Program with a Stack Variable Buffer Overflow

Below this message is information that will help you locate the error. First, we see that this operation is a write to memory (as opposed to a read) of four bytes (which would be an `int` on this system). Second, we see a stack trace, much like one that would be provided by a debugger. The top of the stack (item #0) shows that the error was detected in function `main` at line 11 of file stack.c. The code shown in Figure 1 comes from stack.c and we can see that line 11 is where we are writing to an element of the array.

The rest of the stack trace shows how we reached this line of code, including the various internal functions that were executed as part of starting this program.

```
==2855737==ERROR: AddressSanitizer: stack-buffer-overflow on
address 0x7ffff724e588 at pc 0x55e30042a300 bp 0x7ffff724e500 sp
0x7ffff724e4f0
WRITE of size 4 at 0x7ffff724e588 thread T0
    #0 0x55e30042a2ff in main /home/stack.c:11
    #1 0x7fafcbe80d8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7fafcbe80e3f in __libc_start_main_impl ../csu/libc-
start.c:392
    #3 0x55e30042a144 in _start (/home/stack+0x1144)

Address 0x7ffff724e588 is located in stack of thread T0 at offset
88 in frame
    #0 0x55e30042a218 in main /home/stack.c:6

  This frame has 1 object(s):
    [48, 88) 'stack' (line 7) <== Memory access at offset 88
overflows this variable
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/stack.c:11
in main
```

Figure 2: AddressSanitizer Output for Stack Variable Overflow

You can see that the information provided by AddressSanitizer gives us information that can help us to quickly locate the source of the error. Remember that memory errors are insidious. Sometimes a program will crash immediately when such an error occurs, like this simple example. However, sometimes the program will overwrite some variable that will then cause an erroneous result or crash millions of instructions later in a completely (and unpredictably) different part of the program.

43.5.2 Example: Program with a Global Buffer Overflow

The second example program, shown in Figure 3, is similar to the first one, except that the buffer `glob` is declared outside of any function, so it is global to all functions. It is allocated once and remains until the program terminates (unlike a local variable on the stack that only exists as long as the function is executing). The program is compiled in the same way as the first example to enable AddressSanitizer and then run.

```
04: int glob[10];
05:
06: int main(int argc, char *argv[])
07: {
08:    int i;
09:
10:    for (i=0; i<100; i++)
11:       glob[i] = 7;
12:    exit (0);
13: }
```

Figure 3: Program with a Global Variable Buffer Overflow

Again, when the program was run, AddressSanitizer detected the memory overflow, as shown in Figure 4. As we can see from the first line of output (shown in red), AddressSanitizer knew that this write was to a global variable. The write was identified as being in function `main` on line 11 of the file global.c.

```
==2642571==ERROR: AddressSanitizer: global-buffer-overflow on
address 0x560142d0f0c8 at pc 0x560142d0c29e bp 0x7ffef2c39340 sp
0x7ffef2c39330
WRITE of size 4 at 0x560142d0f0c8 thread T0
    #0 0x560142d0c29d in main /home/global.c:11
    #1 0x7f4801dc6d8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7f4801dc6e3f in __libc_start_main_impl ../csu/libc-
start.c:392
    #3 0x560142d0c164 in _start (/home/global+0x1164)

0x560142d0f0c8 is located 0 bytes to the right of global variable
'glob' defined in 'global.c:4:5' (0x560142d0f0a0) of size 40
SUMMARY: AddressSanitizer: global-buffer-overflow
/home/global.c:11 in main
```

Figure 4: AddressSanitizer Output for Global Variable Overflow

43.5.3 Example: Program with a Heap Buffer Overflow

The third example program, shown in Figure 5, is similar to the first two, except that in this program we are dynamically allocating the memory on the heap. The pointer variable `p` is declared on line 6 and then memory is allocated on line 9 with the address of the allocated memory stored in `p`.

The program is compiled in the same way as the first two examples to enable AddressSanitizer and then run.

```
04: int main (int argc, char *argv[])
05: {
06:    int *p;
07:    int i;
08:
09:    p = (int *)malloc(sizeof(int)*10);
10:    for (i=0; i<100; i++)
11:       p[i] = 7;
12:    exit (0);
13: }
```

Figure 5: Program with a Heap Variable Buffer Overflow

```
==2655076==ERROR: AddressSanitizer: heap-buffer-overflow on
address 0x504000000038 at pc 0x563a713e428e bp 0x7ffeb38a6830 sp
0x7ffeb38a6820
WRITE of size 4 at 0x504000000038 thread T0
    #0 0x563a713e428d in main /home/heap.c:11
    #1 0x7fda9342dd8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7fda9342de3f in __libc_start_main_impl ../csu/libc-
start.c:392
    #3 0x563a713e4144 in _start (/home/heap+0x1144)

0x504000000038 is located 0 bytes to the right of 40-byte region
[0x504000000010,0x504000000038)
allocated by thread T0 here:
    #0 0x7fda936e1887 in __interceptor_malloc
../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x563a713e4234 in main /home/heap.c:9
    #2 0x7fda9342dd8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: heap-buffer-overflow /home/heap.c:11 in
main
```

Figure 6: AddressSanitizer Output for Heap Variable Overflow

In Figure 6, AddressSanitizer detected the memory error, identifying it was an erroneous access to a heap variable. As with the previous examples, the location of the erroneous access was correctly identified.

For heap errors, we also get some extra information about where the variable was allocated. We can see from Figure 6 that the variable was allocated in function `main` on line 9 of heap.c. While the allocation location is pretty obvious in this simple example, in a real complex program with many source files and lines of code, finding the allocating location can be much more difficult.

43.5.4 Example: A Program with a Double Free

A common programming mistake is to free a heap variable after it has already been freed. Such an operation, as shown in Figure 9: Program with a Use after Free Error, can cause corruption of the heap's internal data structures or just cause unpredictable behaviors in a program. The presence of a double free is a sign that the program has some basic implementation problems.

```
05: int main (int argc, char *argv[])
06: {
07:     int *p;
08:     p = (int *)malloc(sizeof(int)*10);
09:     free(p);
10:     free(p);
11: }
```

Figure 7: Program with a Double Free

```
==2633887==ERROR: AddressSanitizer: attempting double-free on
0x604000000090 in thread T0:
    #0 0x55d6ef265eb2 in free (/home/doublefree+0xa0eb2) (BuildId:
a8568d77f87d2a746fa3d3c7add4c3f5b43d849f)
    #1 0x55d6ef2a0ee5 in main /home/doublefree.c:10:5
    #2 0x7fcb7afb1d8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #3 0x7fcb7afb1e3f in __libc_start_main csu/../csu/libc-
start.c:392:3
    #4 0x55d6ef1e3314 in _start (/home/doublefree+0x1e314)
(BuildId: a8568d77f87d2a746fa3d3c7add4c3f5b43d849f)

0x604000000090 is located 0 bytes inside of 40-byte region
[0x604000000090,0x6040000000b8)
freed by thread T0 here:
    #0 0x55d6ef265eb2 in free (/home/doublefree+0xa0eb2) (BuildId:
a8568d77f87d2a746fa3d3c7add4c3f5b43d849f)
    #1 0x55d6ef2a0edc in main /home/doublefree.c:8:16
    #2 0x7fcb7afb1d8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16

previously allocated by thread T0 here:
    #0 0x55d6ef26615e in __interceptor_malloc
(/home/doublefree+0xa115e) (BuildId:
```

```
a8568d77f87d2a746fa3d3c7add4c3f5b43d849f)
    #1 0x55d6ef2a0ecf in main /home/doublefree.c:9:16
    #2 0x7fcb7afb1d8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16

SUMMARY: AddressSanitizer: double-free (/home/doublefree+0xa0eb2)
(BuildId: a8568d77f87d2a746fa3d3c7add4c3f5b43d849f) in free
```

Figure 8: AddressSanitizer Output for Double Free Error

AddressSanitizer is good at detecting such problems as shown by the output in Figure 10: AddressSanitizer Output for Use after Free Error. As in the previous examples, we see that AddressSanitizer identified the cause of the problem and the line on which the second free operation occurred. The second free operation actually occurred in the free function (stack frame #0), which is not very informative. However, in frame #1, see that free was called in function `main` from line 10 in file doublefree.c. (The ":5" part of "10:5" means that the code starts on column 5 of line 10 of the file).

We also see that AddressSanitizer identified where the memory was originally freed, in function `main` on line 9 of file doublefree.c. As for the previous examples, finding the original and offending second-free lines of code in this simple program does not take much effort. However, in a real and complex system, AddressSanitizer can save you hours, if not days of work finding the source of your problem.

43.5.5 Example: A Program with a Use after Free Error

Another common programming mistake is to use a heap variable after it has been freed. If the memory that was freed had not been allocated again, an operation, such as shown in on line 10 of Figure *9*, can cause the reading or writing of free memory. If the memory that was freed had been reallocated, then the read or write operation will be to memory that already has another purpose.

```
05: int main (int argc, char *argv[])
06: {
07:     int *p;
08:     p = (int *)malloc(sizeof(int)*10);
09:     free(p);
10:     *p = 7;
11: }
```

Figure 9: Program with a Use after Free Error

Again, we see that AddressSanitizer is good at detecting such problems as shown by the output in Figure *10*. AddressSanitizer identified the cause of the problem and the line on which the free operation occurred in function `main` on line 9 of useafterfree.c. In addition, it reports the line on which the variable was originally allocated (line 8).

As we discussed previously, finding the allocating code, freeing code, and bad memory reference in this simple program does not take much effort. However, in a real system, the effort can be enormous.

```
==1583248==ERROR: AddressSanitizer: heap-use-after-free on address
0x504000000010 at pc 0x55761311b25c bp 0x7ffc5c6ba7b0 sp
0x7ffc5c6ba7a0
WRITE of size 4 at 0x504000000010 thread T0
    #0 0x55761311b25b in main /home/useafterfree.c:10
    #1 0x7fb5af46dd8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7fb5af46de3f in __libc_start_main_impl ../csu/libc-
start.c:392
    #3 0x55761311b124 in _start (/home/useafterfree+0x1124)

0x504000000010 is located 0 bytes inside of 40-byte region
[0x504000000010,0x504000000038)
freed by thread T0 here:
    #0 0x7fb5af721537 in __interceptor_free
../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127
    #1 0x55761311b224 in main /home/useafterfree.c:9
    #2 0x7fb5af46dd8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58

previously allocated by thread T0 here:
    #0 0x7fb5af721887 in __interceptor_malloc
../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x55761311b214 in main /home/useafterfree.c:8
    #2 0x7fb5af46dd8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: heap-use-after-free
/home/bart/hacks/542/AddressSanitizer/useafterfree.c:10 in main
```

Figure 10: AddressSanitizer Output for Use after Free Error

43.6 Summary

In this chapter, we described how dynamic memory checking tools work and showed how to use a popular such tool, AddressSanitizer, to find common memory errors. Any programmer that has written much C or C++ code has had the experience of trying find and fix memory errors. Tools such as AddressSanitizer make finding these errors significantly easier. And these tools can find errors that are present but are causing errors that have no visible effect. These latent errors are like hidden landmines waiting to be triggered by accident or malicious intent.

43.7 Exercises

1. Run AddressSanitizer on a C or C++ program that you have written or have access to. Did you find any errors that you did not know were present in your code?
2. Research another dynamic memory checking tool beside AddressSanitizer. (Some other tools are listed in Section 43.1.) Learn how to run the chosen tool and try it out on a C or C++ program that you have written or have access to.