**Code Sanitizer**

# Sanitizer

A code sanitizer is a programming tool that detects bugs in the form of undefined or suspicious behavior by a compiler inserting instrumentation code at runtime.

The class of tools was first introduced by Google's AddressSanitizer (or ASan) of 2012, which uses directly mapped shadow memory to detect memory corruption such as buffer overflows or accesses to a dangling pointer (use-after-free).

1. Address Sanitizer
2. Thread Sanitizer
3. Memory Sanitizer
4. Leak Sanitizer
5. Undefined Behavior Sanitizer

# Address Sanitizer

AddressSanitizer (ASan) is a fast memory error detector. It finds use-after-free and {heap,stack,global}-buffer overflow bugs in C/C++ programs. It finds:

* Use after free (dangling pointer dereference)
* Heap buffer overflow
* Stack buffer overflow
* Global buffer overflow
* Use after return
* Use after scope
* Initialization order bugs
* Memory leaks

## Using AddressSanitizer

Compile and link your program with the -fsanitize=address flag.

To get a reasonable performance add -O1 or higher.

To get nicer stack traces in error messages add -fno-omit-frame-pointer.

**Example**

int main() {

int \* pint = new int(5);

delete pint;

return \*pint;

}

Runtime error

====================

==1==ERROR: AddressSanitizer: heap-use-after-free on address 0x602000000010 at pc 0x000000401216 bp 0x7fffe83f0890 sp 0x7fffe83f0888

READ of size 4 at 0x602000000010 thread T0

#0 0x401215 in main /app/example.cpp:4

#1 0x7fb9b6867082 in \_\_libc\_start\_main (/lib/x86\_64-linux-gnu/libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)

#2 0x4010bd in \_start (/app/output.s+0x4010bd) (BuildId: 998b3620e71c714f29f109e702248bafb3f727f3)

0x602000000010 is located 0 bytes inside of 4-byte region [0x602000000010,0x602000000014)

freed by thread T0 here:

#0 0x7fb9b6eea428 in operator delete(void\*, unsigned long) (/opt/compiler-explorer/gcc-13.1.0/lib64/libasan.so.8+0xdc428) (BuildId: c9b24be17e4cbd04bdb4891782c4425e47a9259a)

#1 0x4011de in main /app/example.cpp:3

#2 0x7fb9b6867082 in \_\_libc\_start\_main (/lib/x86\_64-linux-gnu/libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)

previously allocated by thread T0 here:

#0 0x7fb9b6ee9528 in operator new(unsigned long) (/opt/compiler-explorer/gcc-13.1.0/lib64/libasan.so.8+0xdb528) (BuildId: c9b24be17e4cbd04bdb4891782c4425e47a9259a)

#1 0x401187 in main /app/example.cpp:2

#2 0x7fb9b6867082 in \_\_libc\_start\_main (/lib/x86\_64-linux-gnu/libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)

SUMMARY: AddressSanitizer: heap-use-after-free /app/example.cpp:4 in main

Shadow bytes around the buggy address:

0x601ffffffd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0x601ffffffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0x601ffffffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0x601fffffff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0x601fffffff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

=>0x602000000000: fa fa[fd]fa fa fa fa fa fa fa fa fa fa fa fa fa

0x602000000080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa

0x602000000100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa

0x602000000180: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa

0x602000000200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa

0x602000000280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa

Shadow byte legend (one shadow byte represents 8 application bytes):

Addressable: 00

Partially addressable: 01 02 03 04 05 06 07

Heap left redzone: fa

Freed heap region: fd

Stack left redzone: f1

Stack mid redzone: f2

Stack right redzone: f3

Stack after return: f5

Stack use after scope: f8

Global redzone: f9

Global init order: f6

Poisoned by user: f7

Container overflow: fc

Array cookie: ac

Intra object redzone: bb

ASan internal: fe

Left alloca redzone: ca

Right alloca redzone: cb

==1==ABORTING

# Thread Sanitizer

ThreadSanitizer is a fast data race detector for C/C++ and Go.

Data races are one of the most common and hardest to debug types of bugs in concurrent systems.

A data race occurs when two threads access the same variable concurrently and at least one of the accesses is write.

C++11 standard officially bans data races as undefined behavior.

## Using ThreadSanitizer

Simply compile your program with -fsanitize=thread and link it with -fsanitize=thread.

To get a reasonable performance add -O2.

Use -g to get file names and line numbers in the warning messages.

To get nicer stack traces in error messages add -fno-omit-frame-pointer.

**Example**

#include <thread>

int global\_val = 0;

void routine() {

++global\_val;

}

int main() {

std::thread thr(routine);

--global\_val;

thr.join();

return 0;

}

Runtime Error

==================

WARNING: ThreadSanitizer: data race (pid=1)

Read of size 4 at 0x0000004040a4 by main thread:

#0 main /app/example.cpp:11 (output.s+0x4012a9) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

Previous write of size 4 at 0x0000004040a4 by thread T1:

#0 routine() /app/example.cpp:6 (output.s+0x401267) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

#1 void std::\_\_invoke\_impl<void, void (\*)()>(std::\_\_invoke\_other, void (\*&&)()) /opt/compiler-explorer/gcc-13.1.0/include/c++/13.1.0/bits/invoke.h:61 (output.s+0x401e6e) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

#2 std::\_\_invoke\_result<void (\*)()>::type std::\_\_invoke<void (\*)()>(void (\*&&)()) /opt/compiler-explorer/gcc-13.1.0/include/c++/13.1.0/bits/invoke.h:96 (output.s+0x401dd3) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

#3 void std::thread::\_Invoker<std::tuple<void (\*)()> >::\_M\_invoke<0ul>(std::\_Index\_tuple<0ul>) /opt/compiler-explorer/gcc-13.1.0/include/c++/13.1.0/bits/std\_thread.h:292 (output.s+0x401d38) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

#4 std::thread::\_Invoker<std::tuple<void (\*)()> >::operator()() /opt/compiler-explorer/gcc-13.1.0/include/c++/13.1.0/bits/std\_thread.h:299 (output.s+0x401ce2) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

#5 std::thread::\_State\_impl<std::thread::\_Invoker<std::tuple<void (\*)()> > >::\_M\_run() /opt/compiler-explorer/gcc-13.1.0/include/c++/13.1.0/bits/std\_thread.h:244 (output.s+0x401c9c) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

#6 <null> <null> (libstdc++.so.6+0xe8ac2) (BuildId: 60915ee46c169683f422e31f8caeb5d5edd58605)

Location is global 'global\_val' of size 4 at 0x0000004040a4 (output.s+0x4040a4)

Thread T1 (tid=3, finished) created by main thread at:

#0 pthread\_create <null> (libtsan.so.2+0x40cd6) (BuildId: 4e0d0a71dfb217392f9c0b4e6f757e50fa8e9242)

#1 std::thread::\_M\_start\_thread(std::unique\_ptr<std::thread::\_State, std::default\_delete<std::thread::\_State> >, void (\*)()) <null> (libstdc++.so.6+0xe8e3b) (BuildId: 60915ee46c169683f422e31f8caeb5d5edd58605)

#2 main /app/example.cpp:10 (output.s+0x40129f) (BuildId: f46fc74864a6c37575ae6ae8ac4c39230d370c25)

SUMMARY: ThreadSanitizer: data race /app/example.cpp:11 in main

==================

ThreadSanitizer: reported 1 warnings

# Leak Sanitizer

LeakSanitizer is a memory leak detector which is integrated into AddressSanitizer.

The tool is supported on x86\_64 Linux and OS X.

LeakSanitizer is enabled by default in ASan builds of x86\_64 Linux, and can be enabled with ASAN\_OPTIONS=detect\_leaks=1 on x86\_64 OS X.

LSan can also be used without ASan instrumentation.

LSan adds almost no performance overhead until the very end of the process, at which point there is an extra leak detection phase.

## Using LeakSanitizer

To use LSan, simply build your program with AddressSanitizer

**Example**

int main() {

int \* pint = new int(5);

return 0;

}

Runtime Error

================

==1==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:

#0 0x7f262793b528 in operator new(unsigned long) (/opt/compiler-explorer/gcc-13.1.0/lib64/libasan.so.8+0xdb528) (BuildId: c9b24be17e4cbd04bdb4891782c4425e47a9259a)

#1 0x401167 in main /app/example.cpp:2

#2 0x7f26272b9082 in \_\_libc\_start\_main (/lib/x86\_64-linux-gnu/libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)

SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1 allocation(s).

# Memory Sanitizer

MemorySanitizer (MSan) is a detector of uninitialized memory reads in C/C++ programs.

Uninitialized values occur when stack- or heap-allocated memory is read before it is written. MSan detects cases where such values affect program execution.

MSan implements a subset of functionality found in Valgrind (Memcheck tool). It is significantly faster than Memcheck.

## Using MemorySanitizer

To use MSan, compile and link your program with -fsanitize=memory

To get nicer stack traces in error messages add -fno-omit-frame-pointer

**Example**

// NOTE: this program is compiled with clang compiler

int fun() {

int val;

return val;

}

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

int res = fun();

return 0;

}

Runtime Error

==1==WARNING: MemorySanitizer: use-of-uninitialized-value

#0 0x5590b0a145be (/app/output.s+0xa85be)

#1 0x5590b0a146be (/app/output.s+0xa86be)

#2 0x7fd02291d082 (/lib/x86\_64-linux-gnu/libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1dee)

#3 0x5590b098b29d (/app/output.s+0x1f29d)

SUMMARY: MemorySanitizer: use-of-uninitialized-value (/app/output.s+0xa85be)

Exiting

# Undefined Behavior Sanitizer

UndefinedBehaviorSanitizer (UBSan) is a fast undefined behavior detector.

UBSan modifies the program at compile-time to catch various kinds of undefined behavior during program execution, for example:

* Array subscript out of bounds, where the bounds can be statically determined
* Bitwise shifts that are out of bounds for their data type
* Dereferencing misaligned or null pointers
* Signed integer overflow
* Conversion to, from, or between floating-point types which would overflow the destination

## Using UndefinedBehaviorSanitizer

compile and link your program with the -fsanitize=undefined option

**Example**

#include <stdio.h>

int fun() {

int val = 5;

int res = (++val) + (val++);

}

int main() {

int res = fun();

return res;

}

Runtime Error:

/app/example.cpp:3:5: runtime error: execution reached the end of a value-returning function without returning a value

# Hardware-assisted Address Sanitizer (HWASAN)

HWASAN, or Hardware-assisted AddressSanitizer, a newer variant of AddressSanitizer that consumes much less memory.

## Comparison with AddressSanitizer

HWASAN:

* Is less portable than AddressSanitizer as it relies on hardware Address Tagging (AArch64). Address Tagging can be emulated with compiler instrumentation, but it will require the instrumentation to remove the tags before any load or store, which is infeasible in any realistic environment that contains non-instrumented code.
* May have compatibility problems if the target code uses higher pointer bits for other purposes.
* May require changes in the OS kernels (e.g. Linux seems to dislike tagged pointers passed from address space: https://www.kernel.org/doc/Documentation/arm64/tagged-pointers.txt).
* Does not require redzones to detect buffer overflows, but the buffer overflow detection is probabilistic, with roughly 1/(2\*\*TS) chance of missing a bug (6.25% or 0.39% with 4 and 8-bit TS respectively).
* Does not require quarantine to detect heap-use-after-free, or stack-use-after-return. The detection is similarly probabilistic.

The memory overhead of HWASAN is expected to be much smaller than that of AddressSanitizer: 1/TG extra memory for the shadow and some overhead due to TG-aligning all objects.

# References

<https://en.wikipedia.org/wiki/Code_sanitizer>

<https://github.com/google/sanitizers>

<https://github.com/google/sanitizers/wiki>

<https://github.com/google/sanitizers/wiki/AddressSanitizer>

<https://clang.llvm.org/docs/AddressSanitizer.html>

<https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual>

<https://clang.llvm.org/docs/ThreadSanitizer.html>

<https://github.com/google/sanitizers/wiki/MemorySanitizer>

<https://clang.llvm.org/docs/MemorySanitizer.html>

<https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer>

<https://clang.llvm.org/docs/LeakSanitizer.html>

<https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>