



# Debugging with compiler-based feedback

Debugging, Testing and Correctness Workshop Series 2023

Joachim Jenke (jenke@itc.rwth-aachen.de)

Today's materials:

<https://git-ce.rwth-aachen.de/hpc-public/sanitizer-tutorial>

## Examples of Undefined Behavior as Defined in C/C++

---

- Read of uninitialized variable
- Out-of-bounds access to heap, stack and globals
  - Use after free / return / scope
- Bit shifts beyond type bounds
- Dereferencing a NULL pointer
- Signed integer overflow
- Conversion to floating point overflowing the destination
- Accessing a different union member than most recently written
- Data race
- More: see C/C++ standard

## Undefined Behavior: What could go wrong?

- UB allows compilers any behavior
- Possible optimization: assume absence of UB
- Unexpected results
- Avoid UB in any case!

```
int check_for_32(int a) {  
    int res;  
    if (a == 32)  
        res = a;  
    return res;  
}
```

clang 17:

```
check_for_32(int):  
    mov     eax, edi    # return a  
    ret
```

gcc 13:

```
check_for_32(int):  
    mov     eax, 32     # return 32  
    ret
```

- Program is only well-defined, if a=32!
- Both compilers generate valid code!

## First Compiler Feedback

---

- The most important compiler feedback are warnings!
- Compile the code with `-Wall`, `-Wextra`, `-pedantic`, `-Weverything`

```
$ clang -Wall check32.c
```

```
check32.c:5:7: warning: variable 'res' is used uninitialized whenever 'if' condition is false  
[-Wsometimes-uninitialized]
```

```
    if (a == 32)
```

```
        ^~~~~~
```

```
check32.c:7:10: note: uninitialized use occurs here
```

```
    return res;
```

```
        ^~~
```

```
check32.c:5:3: note: remove the 'if' if its condition is always true
```

```
    if (a == 32)
```

```
    ^~~~~~
```

```
check32.c:4:10: note: initialize the variable 'res' to silence this warning
```

```
    int res;
```

```
        ^
```

```
    = 0
```

# Undefined Behavior: What could go wrong?

```
void contains_null_check(int *P) {
    int dead = *P;
    if (P == 0)
        return;
    *P = 4;
}
```

*Dead code elimination* can remove the first statement.  
Will not result in SEGFAULT at runtime.

*Redundant null check elimination* can assume  $P \neq 0$   
→ condition is always false.  
DCE removes all but last statement.

clang 17:

```
contains_null_check(int*):
    test    rdi, rdi                # P == 0
    je      .LBB0_2                 # skip
    mov     dword ptr [rdi], 4      # *P = 4
.LBB0_2:
    ret                                # return
```

gcc 13:

```
contains_null_check(int*):
    mov     DWORD PTR [rdi], 4      # *P = 4
    ret                                # return
```

## Agenda

---

- MemorySanitizer (~10 + 20)
- AddressSanitizer (~15 + 10)
- UBSanitizer (~10 + 10)
- ThreadSanitizer (~20 + 20)

Activate sanitizers with `clang -fsanitize=<name>`

All source codes shown, hands-on codes and these slides are available at:  
<https://git-ce.rwth-aachen.de/hpc-public/sanitizer-tutorial>

For future reference: checkout Durham23

## LLVM Sanitizers Runtime Options

---

- export runtime options as `MSAN_OPTIONS`, `ASAN_OPTIONS` or `TSAN_OPTIONS`
- \$ export `MSAN_OPTIONS=help=1`
- \$ export `TSAN_OPTIONS="history_size=5 log_path=tsan.log"`

# MemorySanitizer

- Detects read of uninitialized memory

```
$ clang -fsanitize=memory -fno-omit-frame-pointer -g -O2 2_msan/msan.c
```

```
int check_for_32(int *a) {
    int res;
    if (*a == 32)
        res = *a;
    return res;
}

int main(){
    int a;
    printf("check_for_32(%i) = %i\n", a, check_for_32(&a) );
}
```

```
==145056==WARNING: MemorySanitizer:
use-of-uninitialized-value
```

```
#0 0x55dbd1ab8571 in check_for_32 msan.c:14:7
#1 0x55dbd1ab8656 in main msan.c:23:40
#2 0x7fa9ccacbd8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16
#3 0x7fa9ccacbe3f in __libc_start_main
csu/../csu/libc-start.c:392:3
#4 0x55dbd1a322a4 in _start (a.out+0x1e2a4)
(BuildId: 763eb80b676294a02a65890f5181a8a697e4b2a5)
```

```
SUMMARY: MemorySanitizer: use-of-uninitialized-value
msan.c:14:7 in check_for_32
Exiting
```



## Conditional Instrumentation

---

- Skip instrumentation of specific function:
  - `__attribute__((no_sanitize("memory")))`
- Skip instrumentation with any sanitizer:
  - `__attribute__((disable_sanitizer_instrumentation))`
- Skip instrumentation based on source file:
  - compile file without MSan flag
- Guard MSan-specific code

```
#if defined(__has_feature)
#  if __has_feature(memory_sanitizer)
// code that builds only under MemorySanitizer
#  endif
#endif
```

```
$ module use ~dc-prot1/.modules  
$ module load clang_comp/17.0.6
```

```
$ git clone https://github.com/UoB-HPC/TeaLeaf.git
```

```
$ cd TeaLeaf
```

```
$ CC=clang CXX=clang++ cmake . -BBUILD-msan -DCMAKE_BUILD_TYPE=Debug \  
-DMODEL=omp -DCMAKE_CXX_FLAGS="-fsanitize=memory -fno-omit-frame-pointer"
```

```
$ cmake --build BUILD-msan
```

```
$ time OMP_NUM_THREADS=8 MSAN_OPTIONS=halt_on_error=0 BUILD-msan/omp-tealeaf
```

**Task:** Fix the issue, rebuild, re-execute

If you don't manage to identify the issue, probably ignore the issue?

```
# Opening tea.out as log file.
==82933==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x5631693c5e11 in read_config(Settings&, State**)
TeaLeaf/driver/parse_config.cpp:60:7
    #1 0x5631693c19ad in main TeaLeaf/driver/main.cpp:110:3
    #2 0x7f646ac89d84 in __libc_start_main (/lib64/libc.so.6+0x3ad84)
    #3 0x56316931239d in _start (TeaLeaf/BUILD-msan/omp-tealeaf+0x3339d)

SUMMARY: MemorySanitizer: use-of-uninitialized-value
TeaLeaf/driver/parse_config.cpp:60:7 in read_config(Settings&, State**)
Exiting
```

Ignore the issue:

- Add `__attribute__((no_sanitize("memory")))` to `read_settings`

Fix:

- The source line information might be a bit misleading
- The uninitialized value is `(*states)[ss].radius`
- Modify `read_states` to initialize the `radius` element independent of geometry

After fixing the initial issue, MSan detects hundreds of potentially false issue for `strlen`:

- exporting `MSAN_OPTIONS=intercept_strlen=0` will suppress these reports

## AddressSanitizer

---

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
  - Enable with: `ASAN_OPTIONS=detect_stack_use_after_return=1` (default on Linux).
  - Disable with: `ASAN_OPTIONS=detect_stack_use_after_return=0`.
- Use-after-scope (clang flag: `-fsanitize-address-use-after-scope`)
- Double-free, invalid free
- Memory leaks (experimental)

## AddressSanitizer Example

```
$ clang++ -fsanitize=address -fno-omit-frame-pointer -g -O2 3_asan/asan.cc
```

```
int main(int argc, char **argv) {  
    int *array = new int[100];  
    delete [] array;  
    return array[argc]; // BOOM  
}
```

```
==176065==ERROR: AddressSanitizer: heap-use-after-free on address  
0x614000000044 at pc 0x563c6909f597 bp 0x7ffe9aa80610 sp 0x7ffe9aa80608
```

```
SUMMARY: AddressSanitizer: heap-use-after-free asan.cc:4:10 in main
```

```
Shadow bytes around the buggy address:
```

```
0x0c287fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
=>0x0c287fff8000: fa fa fa fa fa fa fa fa[fd]fd fd fd fd fd fd fd fd  
0x0c287fff8010: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd  
0x0c287fff8020: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd  
0x0c287fff8030: fd fd fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
```

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

```
Addressable:           00  
Heap left redzone:      fa  
Freed heap region:      fd
```

## AddressSanitizer Example 2

```
$ clang++ -fsanitize=address -fno-omit-frame-pointer -g -O2 3_asan/asan2.cc
```

```
int main(int argc, char **argv) {  
    int *array = new int[99];  
    int tmp = array[argc-2]; // BOOM  
    return tmp; // leaking array  
}
```

```
==334009==ERROR: AddressSanitizer: heap-buffer-overflow on address  
0x61400000003c at pc 0x5561eb60e581 bp 0x7ffc0ad4f0c0 sp 0x7ffc0ad4f0b8
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow asan.cc:3:13 in main
```

```
Shadow bytes around the buggy address:
```

```
0x0c287fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
=>0x0c287fff8000: fa fa fa fa fa fa fa fa[fa]00 00 00 00 00 00 00 00  
0x0c287fff8010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0c287fff8020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0c287fff8030: 00 00 00 00 00 00 00 00 00 04 fa fa fa fa fa fa
```

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

```
Addressable:           00  
Heap left redzone:      fa  
Freed heap region:      fd
```

## Continue Analysis after an Error

---

- Might be useful to limit re-run cycles
- Compile with `-fsanitize-recover=address`
- Execute with `ASAN_OPTIONS=halt_on_error=0`
- Execution state is flawed after an error, so use with caution!

`==334840==ERROR: LeakSanitizer: detected memory leaks`

`Direct leak of 396 byte(s) in 1 object(s) allocated from:`

`#0 0x55685430302d in operator new[](unsigned long) (a.out+0xdc02d)`

`#1 0x556854305521 in main asan2.cc:2:16`

`#2 0x7fc9eed61d8f in __libc_start_call_main`

`csu/../sysdeps/nptl/libc_start_call_main.h:58:16`



```
$ module use ~dc-prot1/.modules  
$ module load clang_comp/17.0.6
```

```
$ git clone https://github.com/UoB-HPC/TeaLeaf.git
```

```
$ cd TeaLeaf
```

```
$ CC=clang CXX=clang++ cmake . -BBUILD-asan -DCMAKE_BUILD_TYPE=Release \  
-DMODEL=omp -DCMAKE_CXX_FLAGS=-fsanitize=address
```

```
$ cmake --build BUILD-asan
```

```
$ time OMP_NUM_THREADS=8 ASAN_OPTIONS=halt_on_error=0 BUILD-asan/omp-tealeaf
```

**Task:** Understand the error messages

What happened to the stack trace? (There should be main!)

Can you get more information?

## ASan Hands-on: Solution

---

- Compiler optimizations drop some frame information
- Add the `-fno-omit-frame-pointer` flag to compilation to avoid this behavior
- Add the debugging flag `-g` to the compilation to get source line information

## UndefinedBehaviorSanitizer

---

- Reports all kinds of UB
  - 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
- See documentation for a full list of available checks

## UndefinedBehaviorSanitizer Example

---

```
$ clang -fsanitize=undefined 4_ubsan/ubsan.c
```

```
int main(int argc, char **argv) {  
    int k = 0x7fffffff;  
    k += argc;  
    return 0;  
}
```

```
$ ./a.out
```

```
$ ./a.out 5
```

```
ubsan.c:3:5: runtime error: signed integer overflow: 2147483646 + 2  
cannot be represented in type 'int'
```

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan.c:3:5 in
```

## UndefinedBehaviorSanitizer Example 2

```
$ clang -fsanitize=undefined 4_ubsan/ubsan2.c
```

```
int foo(int i) {  
    int x[2];  
    x[i] = 12;  
    return x[i];  
}
```

```
int main() {  
    return foo(2);  
}
```

ubsan2.c:3:3: runtime error: index 2 out of bounds for type 'int[2]'

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan2.c:3:3 in

ubsan2.c:4:10: runtime error: index 12 out of bounds for type 'int[2]'

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan2.c:4:10 in

More verbose with -O2

ubsan2.c:3:3: runtime error: index 2 out of bounds for type 'int[2]'

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan2.c:3:3 in

ubsan2.c:3:3: runtime error: store to address 0x7ffcc9fc4058 with  
insufficient space for an object of type 'int'

0x7ffcc9fc4058: note: pointer points here

bc 55 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

^

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan2.c:3:3 in

```
$ module use ~dc-prot1/.modules  
$ module load clang_comp/17.0.6
```

```
$ git clone https://github.com/UoB-HPC/TeaLeaf.git
```

```
$ cd TeaLeaf
```

```
$ CC=clang CXX=clang++ cmake . -BBUILD-ubsan -DCMAKE_BUILD_TYPE=Debug \  
-DMODEL=omp -DCMAKE_CXX_FLAGS=-fsanitize=undefined
```

```
$ cmake --build BUILD-ubsan
```

```
$ time OMP_NUM_THREADS=8 UBSAN_OPTIONS=halt_on_error=0 \  
BUILD-ubsan/omp-tealeaf
```

No UB detected in TeaLeaf

**Task:** Try out the examples in 4\_ubsan

### C++

- The execution of a program contains a **data race** if it contains **two** potentially **concurrent** conflicting actions, at least one of which is **not atomic**, and **neither happens before** the other, [...]. Any such data race results in **undefined behavior**.

### OpenMP

- **Multiple threads** access the same memory **unordered**, at least one thread writes. If a data race occurs then the result of the program is unspecified.
- There is **no benign** data race in C/C++. It is always UB!

## Common Antipattern + Data Race

---

```
program hello
#ifdef _OPENMP
    use omp_lib
#endif
    implicit none
    integer:: nthreads, threadid
!$omp parallel
#ifdef _OPENMP
    nthreads = OMP_GET_NUM_THREADS()
    threadid = OMP_GET_THREAD_NUM()
    if(threadid.eq.0) then
        write(*,*) "Open-MP version with threads = ", nthreads
    endif
#else
    write(*,*) "Serial version "
#endif
!$omp end parallel
end program
```



## Parallel region is not even necessary!

---

```
#ifdef _OPENMP
#include <omp.h>
#endif

int main() {
#ifdef _OPENMP
    printf("Open-MP version with threads = %i\n", omp_get_max_threads());
#else
    printf("Serial version \n");
#endif
}
```

## Data Race vs. Race Condition

- Data race is UB, race condition is not.
- Sprinkling atomics into the code avoids UB, but not necessarily race conditions

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

int main(int argc, char** argv){
    int a = 0, l = 100; double sum = 0;
    if(argc > 1)
        l = atoi(argv[1]);
    #pragma omp parallel num_threads(8) reduction(+:sum)
        while (a < l)
            sum += sin(a++);
    printf("a=%i, sum=%lf\n", a, sum);
}
```

```
#include <stdio.h>
#include <math.h>
#include <atomic>

int main(int argc, char** argv){
    std::atomic<int> a{0}; int l = 100; double sum = 0;
    if(argc > 1)
        l = atoi(argv[1]);
    #pragma omp parallel num_threads(8) reduction(+:sum)
        while (a < l)
            sum += sin(a++);
    printf("a=%i, sum=%lf\n", a.load(), sum);
}
```

## ThreadSanitizer

---

- Detects data race
- Detects lock order inversion (potential deadlocks)
- Many supported OS/architectures:
  - Android aarch64, x86\_64
  - Darwin arm64, x86\_64
  - FreeBSD
  - Linux aarch64, x86\_64, powerpc64, powerpc64le
  - NetBSD
- New and vectorized runtime library introduced with LLVM 15 (halved space and time overhead)

## ThreadSanitizer: Implementation Details

---

- Based on FastTrack algorithm with tracking limited to 4 log entries per 8 bytes of application memory
- New runtime: log entry size reduced from 8 bytes to 4 bytes
  - Stores {w|r}, {atomic}, {size/pos}, tid, epoch
- Vector clock tracks synchronization state

## ThreadSanitizer Example

```
$ clang -fsanitize=thread -g -O2 5_tsan/tsan.c
```

```
include <pthread.h>
```

```
int Global;
```

```
void *Thread1(void *x) {
```

```
    Global = 42;
```

```
    return x;
```

```
}
```

```
int main() {
```

```
    pthread_t t;
```

```
    pthread_create(&t, NULL, Thread1, NULL);
```

```
    Global = 43;
```

```
    pthread_join(t, NULL);
```

```
    return Global;
```

```
}
```

=====

WARNING: ThreadSanitizer: data race (pid=134056)

Write of size 4 at 0x564d0188c258 by main thread:

#0 main tsan.c:10:9 (a.out+0xe78e2)

Previous write of size 4 at 0x564d0188c258 by thread T1:

#0 Thread1 tsan.c:4:9 (a.out+0xe7899)

Location is global 'Global' of size 4 at 0x564d0188c258 (a.out+0x1493258)

Thread T1 (tid=134058, finished) created by main thread at:

#0 pthread\_create tsan\_interceptors\_posix.cpp:1022:234 (a.out+0x60a8d)

#1 main tsan.c:9:2 (a.out+0xe78d3)

SUMMARY: ThreadSanitizer: data race tsan.c:10:9 in main

=====

ThreadSanitizer: reported 1 warnings



- Developed in cooperation with UoUtah, LLNL since 2014
- Injects OpenMP synchronization (and concurrency) into TSan
- Avoids false positive reports
- Shipped with LLVM since 10.0
- Covers latest OpenMP semantics
- Verify that Archer is active: `ARCHER_OPTIONS=verbose=1`
  - Known issue with Ubuntu: the packaging bricks Archer
  - Known issue of TSan with certain HPC applications: increased overhead for 5+ threads



- export as **ARCHER\_OPTIONS=**
  - `verbose={0|1}` for information about archer status
  - `enable={0|1}` for disabling archer
  - `tasking={0|1}` for task-centric analysis
  - `dispatch_fibers={-1|0|n}` for loop-level analysis
  - `ignore_serial={0|1}` to disable analysis for serial code
  - `all_memory={0|1}` to enable analysis of all\_memory dependences
- Strongly encouraged: **TSAN\_OPTIONS=ignore\_noninstrumented\_modules=1**



- R Task-centric analysis (`tasking=1`)
- C Improved loop-level analysis
- R Loop-centric analysis (`dispatch_fibers`)
- C Improved handling of reductions (avoid omission)
- C Detection of data races in vectorized code

### Work in Progress:

- Enabling TSan support in flang compiler
- Upstream all features listed above



## Using Sanitizers with Fortran code

---

- GCC supports most sanitizers (other than MSan)
- Compile Fortran codes with gfortran + sanitizer flags
- Support in flang is possible, but not yet integrated (the clang/17 module on cosma is patched to accept -fsanitize=thread during this week)
- For Archer support, make sure to link the LLVM OpenMP runtime, not GNU runtime:
  - `$ gfortran -lomp -fopenmp -fsanitize=thread test.f90`
  - `$ gfortran -fopenmp -c -fsanitize=thread test.f90`  
`$ clang -fopenmp -fsanitize=thread --gcc-install-dir=<path-to-gfortran> -lgfortran test.o`
  - The latter links the LLVM TSan runtime which has significantly lower runtime overhead

```
$ module use ~dc-prot1/.modules  
$ module load clang_comp/17.0.6
```

```
$ git clone https://github.com/UoB-HPC/TeaLeaf.git
```

```
$ cd TeaLeaf
```

```
$ CC=clang CXX=clang++ cmake . -BBUILD-tsan -DCMAKE_BUILD_TYPE=Debug \  
-DMODEL=omp -DCMAKE_CXX_FLAGS=-fsanitize=thread
```

```
$ cmake --build BUILD-tsan
```

```
$ time OMP_NUM_THREADS=4 TSAN_OPTIONS=halt_on_error=0 BUILD-tsan/omp-tealeaf
```

No data race detected in TeaLeaf

### Tasks:

- Run with `ARCHER_OPTIONS="verbose=1 enable=0"`  
Understand the nature of the false positive reports
- Run with `OMP_NUM_THREADS=8`

## TSan Hands-on 2

---

- Apply TSan to 5\_tsan/omp\_prime.c
  - Basic build: `clang -fopenmp -lm 5_tsan/omp_prime.c`
- Fix the issues