SQUAREHEAD TECHNOLOGY

HOW SANITIZERS WORK

ANDERS SCHAU KNATTEN
SQUAREHEAD

NDC TechTown 2026

Experience?

Original question

- Do I need to rebuild my dependencies for TSan?
- For ASan? UBSan?
- Why?
- Or Else…?

Today

- AddressSanitizer
- ThreadSanitizer
- UndefinedBehaviorSanitizer

Address Sanitizer (ASan)

- Use after free
- Buffer overflows
- Use after return/scope
- Memory leaks
- Container overflow

Use after free

```
struct Sensor {
    double x_pos;
    double y_pos;
    int reading;
};

int read(const Sensor& sensor)
{
    return sensor.reading;
}
```

```
int main()
{
    auto sensor = std::make_unique<Sensor>(1.3, 3.7, 42);
    const Sensor& observer = *sensor;

    sensor.reset();
    return read(observer);
}
```

```
$ ./a.out; echo $?
42
```

With `-O2`:

```
$ ./a.out; echo $?
0
```

On my Mac:

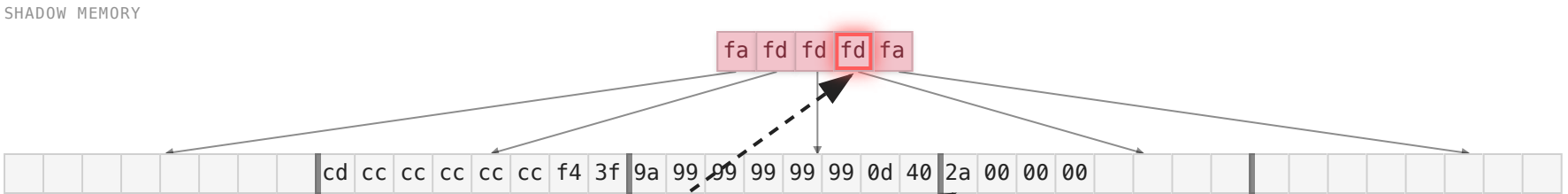
```
$ ./a.out; echo $?
1
```

Memory view


```cpp
struct Sensor {
    double x_pos;
    double y_pos;
    int reading;
};

int read(const Sensor& sensor)
{
    return sensor.reading;
}
```

```cpp
int main()
{
    auto sensor = std::make_unique<Sensor>(1.3, 3.7, 42);
    const Sensor& observer = *sensor;

    sensor.reset();
    return read(observer);
}
```

Shadow memory




```
struct Sensor {
    double x_pos;
    double y_pos;
    int reading;
};

int read(const Sensor& sensor)
{
    return sensor.reading;
}
```

```
int main()
{
    auto sensor = std::make_unique<Sensor>(1.3, 3.7, 42);
    const Sensor& observer = *sensor;

    sensor.reset();
    return read(observer);
}
```

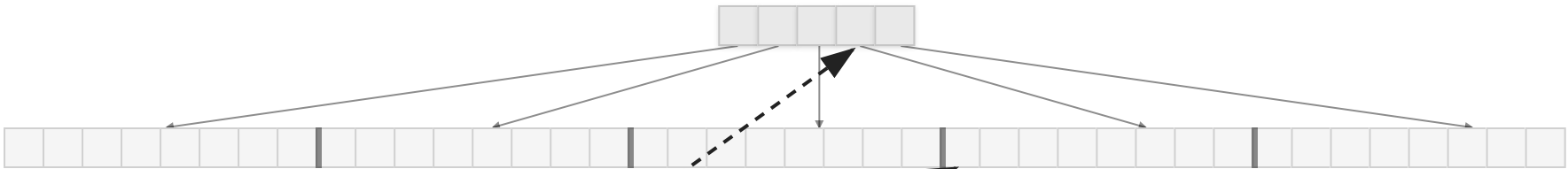
```
==1==ERROR: AddressSanitizer: heap-use-after-free on address 0x75c0981e0050 at pc 0x56fbc0089045 bp 0x7ffc85e91280 sp 0x7
READ of size 4 at 0x75c0981e0050 thread T0
    #0 0x56fbc0089044 in read(Sensor const&) /app/example.cpp:11:17
    #1 0x56fbc00891d5 in main /app/example.cpp:19:10

0x75c0981e0050 is located 16 bytes inside of 24-byte region [0x75c0981e0040,0x75c0981e0058)
freed by thread T0 here:
    #0 0x56fbc0088862 in operator delete(void*, unsigned long) /root/llvm-project/compiler-rt/lib/asan/asan_new_delete.cp
    #1 0x56fbc008994b in std::default_delete<Sensor>::operator()(Sensor*) const /opt/compiler-explorer/gcc-snapshot/lib/g
    #2 0x56fbc00898e3 in std::__uniq_ptr_impl<Sensor, std::default_delete<Sensor>>::reset(Sensor*) /opt/compiler-explorer/
    #3 0x56fbc008948c in std::unique_ptr<Sensor, std::default_delete<Sensor>>::reset(Sensor*) /opt/compiler-explorer/gcc-s
    #4 0x56fbc00891cc in main /app/example.cpp:18:10

previously allocated by thread T0 here:
    #0 0x56fbc0087c0d in operator new(unsigned long) /root/llvm-project/compiler-rt/lib/asan/asan_new_delete.cpp:109:35
    #1 0x56fbc00892af in std::__detail::_MakeUniq<Sensor>::__single_object std::make_unique<Sensor, double, double, int>(
    #2 0x56fbc0089199 in main /app/example.cpp:15:17

SUMMARY: AddressSanitizer: heap-use-after-free /app/example.cpp:11:17 in read(Sensor const&)
Shadow bytes around the buggy address:
  0x75c0981dff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x75c0981dff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x75c0981e0000: fa fa 00 00 00 fa fa fa fd fd[fd]fa fa fa fa fa
  0x75c0981e0080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x75c0981e0100: 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):
  Heap left redzone:       fa
```

Required instrumentation

- Check *every* memory access (gotta go fast!)
- Keep track of *all* allocations / deallocations

Checking memory accesses

SHADOW MEMORY



```
struct Sensor {
    double x_pos;
    double y_pos;
    int reading;
};

int read(const Sensor& sensor)
{
    return sensor.reading;
}
```

```
int main()
{
    auto sensor = std::make_unique<Sensor>(1.3, 3.7, 42);
    const Sensor& observer = *sensor;

    sensor.reset();
    return read(observer);
}
```

Checking memory accesses

`-O2`

```
read(Sensor const&):
    mov     eax, dword ptr [rdi + 16]
    ret
```

`-O2 -fsanitize=address`

```
read(Sensor const&):
    push    rax
    add     rdi, 16
    mov     rax, rdi
    shr     rax, 3                              # 1 byte per 8
    movzx   eax, byte ptr [rax + 2147450880] # + shadow offset
    test    al, al                              # shadow byte = 0?
    jne     .LBB0_1                             # else jump
.LBB0_2:
    mov     eax, dword ptr [rdi]                # normal load
    pop     rcx
    ret                                         # return
.LBB0_1:
    mov     ecx, edi
    and     cl, 7
    add     cl, 3
    cmp     cl, al
    jl      .LBB0_2
    call    __asan_report_load4@PLT
```

Checking memory accesses

- Divide by 8, add an offset
- Happy-case is a single `test` / `jne` fall-through
- Pipelinable, branch predictor friendly, fast! (2x?)
- Code must be instrumented for this to work

Non-instrumented code: Accesses

What if accesses are compiled without `-fsanitize=address`?

```
namespace some_lib {
    int read(const Sensor& sensor)
    {
        return sensor.reading;
    }
}
```

- False negatives. ASan just doesn't check.
- No crashes, false positives, etc

Non-instrumented code: Allocations / deallocations

What if *allocations/deallocations* are compiled without `-fsanitize=address`?

```
namespace some_lib{
    std::unique_ptr<Sensor> get_sensor() {
        return std::make_unique<Sensor>(1.3, 3.7, 42);
    }
}
```

```
int main() {
    auto sensor = some_lib::get_sensor();
    const Sensor& observer = *sensor;

    sensor.reset();
    return read(observer);
}
```

- Allocation happens in non-instrumented code!
- Shadow memory not updated?
- False positives? Crashes?
- No! 🎉

Interceptors

- ASan can intercept and replace `malloc` / `free` / `new` / `delete` at *link time*
- Both for static / dynamic linking
- Implementation differs between
  - `malloc` / `free` vs. `new` / `delete`
  - Static vs. dynamic linking
  - Mac vs. Linux vs. Windows
  - Even gcc vs. Clang on the same platform!
  - Example: `malloc` Linux (ELF)

Interceptors

GCC on Linux, dynamic linking

main (`-fsanitize=address`)

```
int main() {
  dynlib::f();
}
```

libdynlib.so

```
void f() {
  malloc();
}
```

ASan runtime (`libasan.so`)

`malloc` · `free` `new` · `delete` · `...`

```
$ LD_DEBUG=bindings ./build/gcc/main 2>&1 | grep 'libdynlib.so.*to.*malloc'
 29: binding file /w/build/gcc/libdynlib.so [0] to /usr/local/lib64/libasan.so.8 [0]: normal symbol `malloc' [GLIBC_2.17]
```

Interceptors

Clang on Linux

main (`-fsanitize=address`)

```
int main() {
  dynlib::f();
}
```

libdynlib.so

```
void f() {
  malloc();
}
```

ASan runtime

`malloc` · `free` `new` · `delete` · `...`

```
$ LD_DEBUG=bindings ./build/clang/main 2>&1 | grep 'libdynlib.so.*to.*malloc'
  26:   binding file /w/build/clang/libdynlib.so [0] to ./build/clang/main [0]: normal symbol `malloc' [GLIBC_2.17]
```

Load order

```
$ ldd build/gcc/main
  linux-vdso.so.1
  libasan.so.8 => /usr/local/lib64/libasan.so.8
  libdynlib.so => /w/build/gcc/libdynlib.so
  libstdc++.so.6 => /usr/local/lib64/libstdc++.so.6
  libm.so.6 => /lib/aarch64-linux-gnu/libm.so.6
  libgcc_s.so.1 => /usr/local/lib64/libgcc_s.so.1
  libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6
  /lib/ld-linux-aarch64.so.1
```

```
$ ldd build/clang/main
  linux-vdso.so.1
  libdynlib.so => /w/build/clang/libdynlib.so
  libstdc++.so.6 => /usr/local/lib64/libstdc++.so.6
  libm.so.6 => /lib/aarch64-linux-gnu/libm.so.6
  libresolv.so.2 => /lib/aarch64-linux-gnu/libresolv.so.2
  libgcc_s.so.1 => /usr/local/lib64/libgcc_s.so.1
  libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6
  /lib/ld-linux-aarch64.so.1
```

Heap buffer overflow

```cpp
int main() {
    int* buf = new int[2]{0,1};
    int result = buf[2];
    delete[] buf;
    return result;
}
```

```
==1==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x74f02cbe0018 at pc 0x595a5818a0c4 bp 0x7ffcd1f0a450 sp 0x
READ of size 4 at 0x74f02cbe0018 thread T0
    #0 0x595a5818a0c3 in main /app/example.cpp:3:16

0x74f02cbe0018 is located 0 bytes after 8-byte region [0x74f02cbe0010,0x74f02cbe0018)
allocated by thread T0 here:
    #0 0x595a58188d0d in operator new[](unsigned long) /root/llvm-project/compiler-rt/lib/asan/asan_new_delete.cpp:111:37
    #1 0x595a5818a008 in main /app/example.cpp:2:14
(...)
```

- Needs instrumentation for checks, allocations tracked by interceptors
- Worst case: false negatives
- Uses redzones on each side of objects

Stack buffer overflow

```
int main() {
    int buf[2] {0,1};
    return buf[2];
}
```

```
==1==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x740cf64f0028 at pc 0x5d2e49fa20f9 bp 0x7ffe5eddb7d0 sp 0x
READ of size 4 at 0x740cf64f0028 thread T0
    #0 0x5d2e49fa20f8 in main /app/example.cpp:3:10

Address 0x740cf64f0028 is located in stack of thread T0 at offset 40 in frame
    #0 0x5d2e49fa1fff in main /app/example.cpp:1

  This frame has 1 object(s):
    [32, 40) 'buf' (line 2) <== Memory access at offset 40 overflows this variable
```

- Needs instrumentation for checks
- Intercept allocations for tracking? No, there are no allocations.
- Shadow memory updated through instrumentation on function entry/exit
- Still, worst case: false negatives (for normal builds)

Stack use after return

```
int* f() {
    int i = 42;
    return &i;
}

int main() {
    int* ptr = f();
    return *ptr;
}
```

```
==1==ERROR: AddressSanitizer: stack-use-after-return on address 0x730c126f0020 at pc 0x636c5318e07d bp 0x7ffea6a43270 sp (
READ of size 4 at 0x730c126f0020 thread T0
    #0 0x636c5318e07c in main /app/example.cpp:8:10

Address 0x730c126f0020 is located in stack of thread T0 at offset 32 in frame
    #0 0x636c5318df2f in f() /app/example.cpp:1
```

- Needs instrumentation for checks
- Again no allocations to intercept
- Shadow memory updated through instrumentation on function entry/exit

Leak detection

```cpp
int main() {
    int* ptr = new int{42};
    return *ptr;
}
```

```
==1==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x5c43f9422b2d in operator new(unsigned long) /root/llvm-project/compiler-rt/lib/asan/asan_new_delete.cpp:109:35
    #1 0x5c43f9423f38 in main /app/example.cpp:2:14

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

- All allocations/deallocations are intercepted, no instrumentation needed!
- At program exit, check all remaining unreachable allocations
- Again, uninstrumented code doesn't cause false postives

Container bounds

```cpp
int main() {
    std::vector<int> v;
    v.reserve(10);
    v.push_back(42);
    return v[1];
}
```

```
==1==ERROR: AddressSanitizer: container-overflow on address 0x7893969e0054 at pc 0x5947a7b5a0cd bp 0x7ffcf590dff0 sp 0x7f
READ of size 4 at 0x7893969e0054 thread T0
  (...)

0x7893969e0054 is located 4 bytes inside of 40-byte region [0x7893969e0050,0x7893969e0078)
allocated by thread T0 here:
  (...)
```

- Not a buffer overflow! Memory is allocated.
- Shadow memory says ok, needs instrumentation by `std::vector` itself
- `#ifdef ASAN_CONTAINER_BOUNDS_STUFF do_some_extra_stuff()`
- False positive: `push_back` in non-instrumented code, access in instrumented

ASan

Works fine with non-instrumented libraries!

No false positives or crashes, only false negatives.

(Except container bounds, easy to disable, rare?)

ThreadSanitizer (TSan)

- Data races
- Potential deadlocks

A data race

> 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

```
int value = 0;
```

```
void write(int v) {
    value = v;
}
```

```
int read() {
    return value;
}
```

A data race

```
int value = 0;
```

SHADOW STATE

T1 W

T1

```
void write(int v) {
    value = v;
}
```

T2

```
int read() {
    return value;
}
```

```
WARNING: ThreadSanitizer: data race (pid=2)
  Read of size 4 at 0x5555569fa818 by thread T2:
    #0 read() /app/example.cpp:6:10 (output.s+0xea97c)

  Previous write of size 4 at 0x5555569fa818 by thread T1:
    #0 write(int) /app/example.cpp:10:9 (output.s+0xea9cb)

  Location is global 'value' of size 4 at 0x5555569fa818 (output.s+0x14a6818)

  Thread T2 (tid=5, running) created by main thread at:
    #4 main /app/example.cpp:16:16 (output.s+0xeaa33)
```

Atomics

```
std::atomic<bool> ready = false;
```

```
void write(bool r) {
    ready = r;
}
```

```
bool read() {
    return ready;
}
```

Atomics

```
std::atomic<bool> ready = false;
```

SHADOW STATE

T1 W A T2 R A

T1

```
void write(bool r) {
    ready = r;
}
```

T2

```
bool read() {
    return ready;
}
```

Happens-before

```
int value = 0;
std::atomic<bool> ready = false;
```

```
void write(int v) {
    value = v;
    ready = true;
}
```

```
int read() {
    if (!ready) {
        return 0;
    }
    return value;
}
```

The Two Memory Models

Happens-before

```
int value = 0;
std::atomic<bool> ready = false;
```

SHADOW STATE

T1 W @1	T2 R @1
T1 W A @1	T2 R A @1

T1

```
void write(int v) {
    value = v;
    ready = true;
}
```

VECTOR CLOCK

T1:2 T2:0

T2

```
int read() {
    if (!ready) {
        return 0;
    }
    return value;
}
```

VECTOR CLOCK

T1:1 T2:1

Doesn’t happen-before

```
int value = 0;
std::atomic<bool> ready = false;
```

SHADOW STATE

T1 W @2

T1 W A @1 **T2** R A @1

T1

```
void write(int v) {
    ready = true;
    value = v;
}
```

VECTOR CLOCK

T1:2 **T2**:0

T2

```
int read() {
    if (!ready) {
        return 0;
    }
    return value;
}
```

VECTOR CLOCK

T1:1 **T2**:1

```
WARNING: ThreadSanitizer: data race (pid=2)
  Read of size 4 at 0x5555569fa818 by thread T2:
    #0 read() /app/example.cpp:8:10 (output.s+0xea97c)

  Previous write of size 4 at 0x5555569fa818 by thread T1:
    #0 write(int) /app/example.cpp:12:9 (output.s+0xea9cb)
```

Uninstrumented code

```
int value = 0;
std::atomic<bool> ready = false;   // in third_party
```

SHADOW STATE

T1 W @1

T1

```
void write(int v) {
    value = v;
    third_party::mark_ready();
}
```

VECTOR CLOCK

T1:1 **T2**:0

T2

```
int read() {
    if (!third_party::is_ready()) {
        return 0;
    }
    return value;
}
```

VECTOR CLOCK

T1:0 **T2**:1

```
WARNING: ThreadSanitizer: data race (pid=145)
  Read of size 4 at 0xaaaac8cc3968 by thread T2:
    (...)
```

Deadlock detection

```cpp
int main() {
    std::mutex a;
    std::mutex b;
    {
        std::lock_guard guard_a(a);
        std::lock_guard guard_b(b);
    }
    {
        std::lock_guard guard_b(b);
        std::lock_guard guard_a(a);
    }
}
```

- Uses interceptors for mutex locks
- Can also work with uninstrumented libraries! (True positives)
- No false positives

```
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=2)
  Cycle in lock order graph: M0 (0x7fffffffe9c8) => M1 (0x7fffffffe9a0) => M0

  Mutex M1 acquired here while holding mutex M0 in main thread:
    #0 pthread_mutex_lock /root/llvm-project/compiler-rt/lib/tsan/rtl/tsan_interceptors_posix.cpp:1430:3 (output.s+0x66e5
    (...)
    #4 main /app/example.cpp:8:25 (output.s+0xe985b)

  Mutex M0 acquired here while holding mutex M1 in main thread:
    #0 pthread_mutex_lock /root/llvm-project/compiler-rt/lib/tsan/rtl/tsan_interceptors_posix.cpp:1430:3 (output.s+0x66e5
```

TSan

Must rebuild all code (except stdlib) or risk false positives

Can use suppressions

UndefinedBehaviorSanitizer (UBSan)

- Use after free (ASan)?
- Data race (TSan)?

- Signed integer overflow
- Missing return
- Invalid shifts
- Bools that are neither true nor false(!)
- (…)

- Only five minutes left! 😬

UndefinedBehaviorSanitizer (UBSan)

We've seen:

- Local instrumentation
- Global interceptors
- False positives due to uninstrumented code when interceptors aren't enough (e.g. atomics with TSan)

- UBSan: Local reasoning only! No spooky action at a distance
- UBSan: No false positives from uninstrumented code

UBSan example

```
int read_width()  { return 100'000; }
int read_height() { return 100'000; }

int main() {
  const int width = read_width();
  const int height = read_height();

  int64_t area = width * height;

  std::cout << area;
}
```

```
1410065408
```

UBSan example

`-fsanitize=undefined`

```
int read_width()  { return 100'000; }
int read_height() { return 100'000; }

int main() {
  const int width = read_width();
  const int height = read_height();

  int64_t area = width * height;

  std::cout << area;
}
```

```
/app/example.cpp:13:24: runtime error: signed integer overflow: 100000 * 100000 cannot be represented in type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior /app/example.cpp:13:24
1410065408
```

Summary

AddressSanitizer

- Instrumenting accesses
- Intercepting allocations/deallocations
- Doesn’t require rebuilding dependencies

ThreadSanitizer

- Instrumenting accesses
- Instrumenting atomics, intercepting mutexes
- Requires rebuilding dependencies (or use suppressions)

UndefinedBehaviorSanitizer

- Instrumenting operations
- No interceptors
- Doesn’t require rebuilding dependencies

SQUAREHEAD TECHNOLOGY

HOW SANITIZERS WORK

ANDERS SCHAU KNATTEN
SQUAREHEAD

NDC TechTown 2026