# Vulnerability Detection - Sanitizers

## Holistic Software Security

Aravind Machiry

# Bug Manifestations

- How bugs affect program behavior?
  - If we have exhaustive test cases:
    - Actual output != Expected output.
  - In the absence of test cases, i.e., Fuzzing:
    - Memory errors: Program Crashes (SIGSEGV) => Access/Execute invalid memory.
    - There could be bugs which do not result in SIGSEGV.

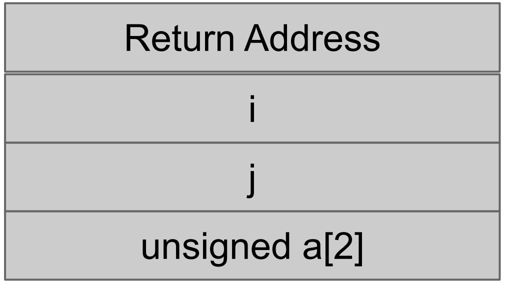
# Silent Bugs

```
int main() {
  unsigned i, j, a[2];
  scanf("%u %u", &i, &j);
  a[i] = j;
  ...
  return 0;
}
```

Here, if i == 2 (off by one error), the program may not crash?

- Why?

**Runtime Stack**

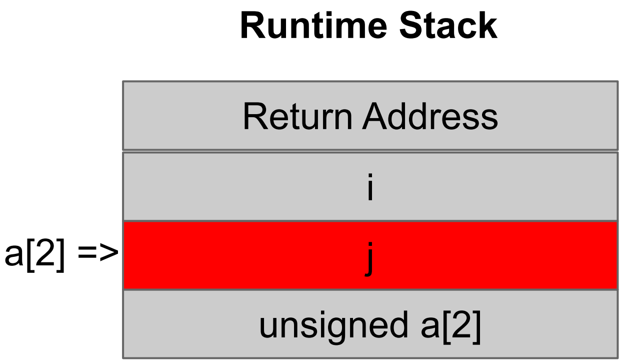


# Silent Bugs

```
int main() {
  unsigned i, j, a[2];
  scanf("%u %u", &i, &j);
  a[i] = j;
  ...
  return 0;
}
```

Here, if i == 2 (off by one error), the program may not crash?

- Why?



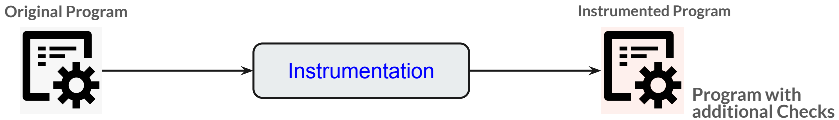
# Improving bug detection

- The behavior of a bug, especially memory corruption, depends on the program state and execution environment.
- Can we detect these bugs without relying on program state?
    - Fuzzing: we detect a bug if it results in the program crash (SIGSEGV).
    - Idea: Make all bugs result in program crashes.

# Sanitizers

- Change the program such that ***we detect bugs when they occur*** instead of waiting for the bugs to result in crash.
- Mechanism: Instrument the program by adding additional checks for detecting bugs.

# Sanitizers: Overview




Original Program

```
int main() {
  unsigned i, j, a[2];
  scanf("%u %u", &i, &j);
  a[i] = j;
  ...
  return 0;
}
```

Array out-of-bounds Sanitizer

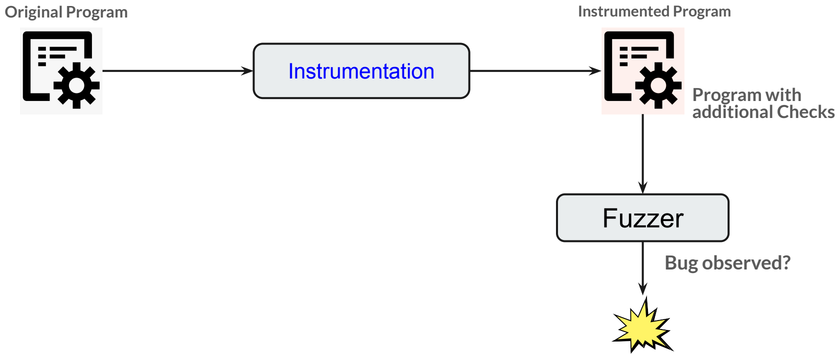
```
int main() {
  unsigned i, j, a[2];
  scanf("%u %u", &i, &j);
  if (i < 2) {
    a[i] = j;
  } else { CRASH}
  ...
  return 0;
}
```

Instrumented Program

# Real world Sanitizers

- Usually bug specific. Examples:
  - MemorySanitizer: Detects *uninitialized reads.*
  - AddressSanitizer: Detects *invalid memory accesses.*
- **General instrumentation idea**: At all instructions in the program where the bug can occur, add a check to detect the bug.
  - AddressSanitizer: Detects Invalid Memory Accesses.
    - Invalid Memory access can occur at load and store instructions.
      - **Instrument every load and store to check if the used address is invalid (i.e., does not belong to a program object).**

# Sanitizers: Usage




Original Program

```
int main() {
  unsigned i, j, a[2];
  scanf("%u %u", &i, &j);
  a[i] = j;
  ...
  return 0;
}
```

Array out-of-bounds Sanitizer

```
int main() {
  unsigned i, j, a[2];
  scanf("%u %u", &i, &j);
  if (i < 2) {
    a[i] = j;
  } else { CRASH}
  ...
  return 0;
}
```

Instrumented Program

# Why can't we always use sanitizers?

They detect bugs at runtime => Why can't we just use sanitizers and not worry about bugs, as they will never lead to vulnerabilities.


Sanitizers introduce **a lot** of overhead.

# Sanitizers Implementation

- Sanitizers need to maintain lot of additional state to check for the possibility of bugs.
    - AddressSanitizer: Detects Invalid Memory Accesses:
        - Need to **maintain metadata** regarding which memory (i.e., address) is valid v/s invalid.
        - Tricky: Handling dynamic memory allocation.

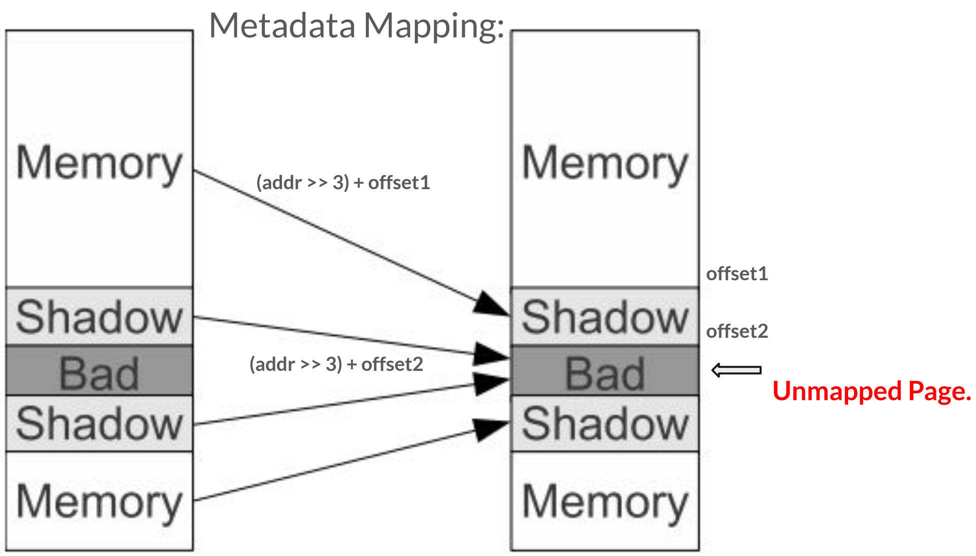
**Popular research direction**: Smart and efficient way to maintain metadata.

# AddressSanitizer (ASan)

- Metadata (or shadow memory):
  - One eighth of the virtual memory will be used to maintain metadata:
    - One bit of metadata for each byte of application memory.
    - Bit is zero: The corresponding address is valid else invalid.
- Accessing metadata for a given address (Addr):
  - Direct Mapping:

```
// Checking 8-byte access
MetadataAddr = (Addr >> 3) + Offset;
if (*MetadataAddr != 0)
  ReportAndCrash(Addr);
```

# ASan: Mapping



# ASan: Usage

a.c

```
void foo(T *a) {
*a = 0x1234;
}
```

clang -fsanitize=address a.c -c -DT=long

```
push %rax
mov %rdi,%rax
shr $0x3,%rax
mov $0x100000000000,%rcx
or %rax,%rcx
cmpb $0x0,(%rcx) # Compare Shadow with 0
jne 23 <foo+0x23> # To Error
movq $0x1234,(%rdi) # Original store
pop %rax
retq
callq __asan_report_store8 # Error
```

# ASan: Conclusion

- One of the most popular sanitizers: Used extensively in fuzzing.

- Overhead:
  - Adds additional instructions:
    - Memory overhead: ~3X (Consumes thrice the amount of memory).
    - Slowdown: ~2X (Runs at half the speed).

# ThreadSanitizer

- Detects data races.
- Where can data races happen i.e., which instructions it should track?
- How to detect a data race? What metadata should be maintained?

# Other sanitizers (supported by clang)

- -fsanitize=address: AddressSanitizer, a memory error detector.
- -fsanitize=thread: ThreadSanitizer, a data race detector.
- -fsanitize=memory: MemorySanitizer, a detector of uninitialized reads. Requires instrumentation of all program code.
- -fsanitize=undefined: UndefinedBehaviorSanitizer, a fast and compatible undefined behavior checker.
- -fsanitize=dataflow: DataFlowSanitizer, a general data flow analysis.

# Sanitizers: Final Thoughts

- They increase the ability of fuzzing to find bugs.
- Always use them with fuzzers: Performance impact does not matter much - lets throw more machines.
- New sanitizers => Always appreciated and could have a high impact.
- Decreasing overhead of sanitizers: Appreciated but may have less impact.