/address-sanitizer

Source: `~/.claude/skills/tob-testing-handbook-skills/skills/address-sanitizer/SKILL.md`

---

name: address-sanitizer type: technique description: > AddressSanitizer detects memory errors during fuzzing. Use when fuzzing C/C++ code to find buffer overflows and use-after-free bugs.

AddressSanitizer (ASan)

AddressSanitizer (ASan) is a widely adopted memory error detection tool used extensively during software testing, particularly fuzzing. It helps detect memory corruption bugs that might otherwise go unnoticed, such as buffer overflows, use-after-free errors, and other memory safety violations.

Overview

ASan is a standard practice in fuzzing due to its effectiveness in identifying memory vulnerabilities. It instruments code at compile time to track memory allocations and accesses, detecting illegal operations at runtime.

Key Concepts

Concept	Description
Instrumentation	ASan adds runtime checks to memory operations during compilation
Shadow Memory	Maps 20TB of virtual memory to track allocation state
Performance Cost	Approximately 2-4x slowdown compared to non-instrumented code
Detection Scope	Finds buffer overflows, use-after-free, double-free, and memory leaks

When to Apply

Apply this technique when:

- Fuzzing C/C++ code for memory safety vulnerabilities
- Testing Rust code with unsafe blocks
- Debugging crashes related to memory corruption
- Running unit tests where memory errors are suspected

Skip this technique when:

- Running production code (ASan can reduce security)
- Platform is Windows or macOS (limited ASan support)
- Performance overhead is unacceptable for your use case
- Fuzzing pure safe languages without FFI (e.g., pure Go, pure Java)

Quick Reference

Task	Command/Pattern
Enable ASan (Clang/GCC)	`-fsanitize=address`
Enable verbosity	`ASAN_OPTIONS=verbosity=1`
Disable leak detection	`ASAN_OPTIONS=detect_leaks=0`
Force abort on error	`ASAN_OPTIONS=abort_on_error=1`
Multiple options	`ASAN_OPTIONS=verbosity=1:abort_on_error=1`

Step-by-Step

Step 1: Compile with ASan

Compile and link your code with the `-fsanitize=address` flag:

```
clang -fsanitize=address -g -o my_program my_program.c
```

The `-g` flag is recommended to get better stack traces when ASan detects errors.

Step 2: Configure ASan Options

Set the `ASAN_OPTIONS` environment variable to configure ASan behavior:

```
export ASAN_OPTIONS=verbosity=1:abort_on_error=1:detect_leaks=0
```

Step 3: Run Your Program

Execute the ASan-instrumented binary. When memory errors are detected, ASan will print detailed reports:

```
./my_program
```

Step 4: Adjust Fuzzer Memory Limits

ASan requires approximately 20TB of virtual memory. Disable fuzzer memory restrictions:

- libFuzzer: `-rss_limit_mb=0`
- AFL++: `-m none`

Common Patterns

Pattern: Basic ASan Integration

Use Case: Standard fuzzing setup with ASan

Before:

```
clang -o fuzz_target fuzz_target.c
./fuzz_target
```

After:

```
clang -fsanitize=address -g -o fuzz_target fuzz_target.c
ASAN_OPTIONS=verbosity=1:abort_on_error=1 ./fuzz_target
```

Pattern: ASan with Unit Tests

Use Case: Enable ASan for unit test suite

Before:

```
gcc -o test_suite test_suite.c -lcheck
./test_suite
```

After:

```
gcc -fsanitize=address -g -o test_suite test_suite.c -lcheck
ASAN_OPTIONS=detect_leaks=1 ./test_suite
```

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Use `-g` flag	Provides detailed stack traces for debugging
Set `verbosity=1`	Confirms ASan is enabled before program starts
Disable leaks during fuzzing	Leak detection doesn't cause immediate crashes, clutters output
Enable `abort_on_error=1`	Some fuzzers require `abort()` instead of `_exit()`

Understanding ASan Reports

When ASan detects a memory error, it prints a detailed report including:

- **Error type**: Buffer overflow, use-after-free, etc.

- **Stack trace**: Where the error occurred
- **Allocation/deallocation traces**: Where memory was allocated/freed
- **Memory map**: Shadow memory state around the error

Example ASan report:

```
==12345==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000eff4 at pc
0x00000048e6a3
READ of size 4 at 0x60300000eff4 thread T0
    #0 0x48e6a2 in main /path/to/file.c:42
```

Combining Sanitizers

ASan can be combined with other sanitizers for comprehensive detection:

```
clang -fsanitize=address,undefined -g -o fuzz_target fuzz_target.c
```

Platform-Specific Considerations

Linux: Full ASan support with best performance **macOS**: Limited support, some features may not work **Windows**: Experimental support, not recommended for production fuzzing

Anti-Patterns

Anti-Pattern	Problem	Correct Approach
Using ASan in production	Can make applications less secure	Use ASan only for testing
Not disabling memory limits	Fuzzer may kill process due to 20TB virtual memory	Set `-rss_limit_mb=0` or `-m none`
Ignoring leak reports	Memory leaks indicate resource management issues	Review leak reports at end of fuzzing campaign

Tool-Specific Guidance

libFuzzer

Compile with both fuzzer and address sanitizer:

```
clang++ -fsanitize=fuzzer,address -g harness.cc -o fuzz
```

Run with unlimited RSS:

```
./fuzz -rss_limit_mb=0
```

Integration tips:

- Always combine `-fsanitize=fuzzer` with `-fsanitize=address`
- Use `-g` for detailed stack traces in crash reports
- Consider `ASAN_OPTIONS=abort_on_error=1` for better crash handling

See: libFuzzer: AddressSanitizer

AFL++

Use the `AFL_USE_ASAN` environment variable:

```
AFL_USE_ASAN=1 afl-clang-fast++ -g harness.cc -o fuzz
```

Run with unlimited memory:

```
afl-fuzz -m none -i input_dir -o output_dir ./fuzz
```

Integration tips:

- `AFL_USE_ASAN=1` automatically adds proper compilation flags
- Use `-m none` to disable AFL++'s memory limit
- Consider `AFL_MAP_SIZE` for programs with large coverage maps

See: AFL++: AddressSanitizer

cargo-fuzz (Rust)

Use the `--sanitizer=address` flag:

```
cargo fuzz run fuzz_target --sanitizer=address
```

Or configure in `fuzz/Cargo.toml`:

```
[profile.release]
opt-level = 3
```

```
debug = true
```

Integration tips:

- ASan is useful for fuzzing unsafe Rust code or FFI boundaries
- Safe Rust code may not benefit as much (compiler already prevents many errors)
- Focus on unsafe blocks, raw pointers, and C library bindings

See: cargo-fuzz: AddressSanitizer

honggfuzz

Compile with ASan and link with honggfuzz:

```
honggfuzz -i input_dir -o output_dir -- ./fuzz_target_asan
```

Compile the target:

```
hfuzz-clang -fsanitize=address -g target.c -o fuzz_target_asan
```

Integration tips:

- honggfuzz works well with ASan out of the box
- Use feedback-driven mode for better coverage with sanitizers
- Monitor memory usage, as ASan increases memory footprint

Troubleshooting

Issue	Cause	Solution
Fuzzer kills process immediately	Memory limit too low for ASan's 20TB virtual memory	Use `-rss_limit_mb=0` (libFuzzer) or `-m none` (AFL++)
"ASan runtime not initialized"	Wrong linking order or missing runtime	Ensure `-fsanitize=address` used in both compile and link
Leak reports clutter output	LeakSanitizer enabled by default	Set `ASAN_OPTIONS=detect_leaks=0`
Poor performance (>4x slowdown)	Debug mode or unoptimized build	Compile with `-O2` or `-O3` alongside `-fsanitize=address`
ASan not detecting obvious bugs	Binary not instrumented	Check with `ASAN_OPTIONS=verbosity=1` that ASan prints startup info
False positives	Interceptor conflicts	Check ASan FAQ for known issues with specific libraries

Related Skills

Tools That Use This Technique

Skill	How It Applies
libfuzzer	Compile with `-fsanitize=fuzzer,address` for integrated fuzzing with memory error detection
aflpp	Use `AFL_USE_ASAN=1` environment variable during compilation
cargo-fuzz	Use `--sanitizer=address` flag to enable ASan for Rust fuzz targets
honggfuzz	Compile target with `-fsanitize=address` for ASan-instrumented fuzzing

Related Techniques

Skill	Relationship
undefined-behavior-sanitizer	Often used together with ASan for comprehensive bug detection (undefined behavior + memory errors)
fuzz-harness-writing	Harnesses must be designed to handle ASan-detected crashes and avoid false positives
coverage-analysis	Coverage-guided fuzzing helps trigger code paths where ASan can detect memory errors

Resources

Key External Resources

AddressSanitizer on Google Sanitizers Wiki

The official ASan documentation covers:

- Algorithm and implementation details
- Complete list of detected error types
- Performance characteristics and overhead
- Platform-specific behavior
- Known limitations and incompatibilities

SanitizerCommonFlags

Common configuration flags shared across all sanitizers:

- `verbosity`: Control diagnostic output level
- `log_path`: Redirect sanitizer output to files
- `symbolize`: Enable/disable symbol resolution in reports
- `external_symbolizer_path`: Use custom symbolizer

AddressSanitizerFlags

ASan-specific configuration options:

- `detect_leaks`: Control memory leak detection
- `abort_on_error`: Call `abort()` vs `_exit()` on error
- `detect_stack_use_after_return`: Detect stack use-after-return bugs
- `check_initialization_order`: Find initialization order bugs

AddressSanitizer FAQ

Common pitfalls and solutions:

- Linking order issues
- Conflicts with other tools
- Platform-specific problems
- Performance tuning tips

Clang AddressSanitizer Documentation

Clang-specific guidance:

- Compilation flags and options
- Interaction with other Clang features
- Supported platforms and architectures

GCC Instrumentation Options

GCC-specific ASan documentation:

- GCC-specific flags and behavior
- Differences from Clang implementation
- Platform support in GCC

AddressSanitizer: A Fast Address Sanity Checker (USENIX Paper)

Original research paper with technical details:

- Shadow memory algorithm
- Virtual memory requirements (historically 16TB, now ~20TB)
- Performance benchmarks
- Design decisions and tradeoffs

Revision #5
Created 2026-02-18 08:40:09 UTC by John
Updated 2026-06-21 20:01:19 UTC by John