/libfuzzer

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

---

name: libfuzzer type: fuzzer description: > Coverage-guided fuzzer built into LLVM for C/C++ projects. Use for fuzzing C/C++ code that can be compiled with Clang.

libFuzzer

libFuzzer is an in-process, coverage-guided fuzzer that is part of the LLVM project. It's the recommended starting point for fuzzing C/C++ projects due to its simplicity and integration with the LLVM toolchain. While libFuzzer has been in maintenance-only mode since late 2022, it is easier to install and use than its alternatives, has wide support, and will be maintained for the foreseeable future.

When to Use

Fuzzer	Best For	Complexity
libFuzzer	Quick setup, single-project fuzzing	Low
AFL++	Multi-core fuzzing, diverse mutations	Medium

Fuzzer	Best For	Complexity
LibAFL	Custom fuzzers, research projects	High
Honggfuzz	Hardware-based coverage	Medium

Choose libFuzzer when:

- You need a simple, quick setup for C/C++ code
- Project uses Clang for compilation
- Single-core fuzzing is sufficient initially
- Transitioning to AFL++ later is an option (harnesses are compatible)

Note: Fuzzing harnesses written for libFuzzer are compatible with AFL++, making it easy to transition if you need more advanced features like better multi-core support.

Quick Start

```
#include <stdint.h>
#include <stddef.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Validate input if needed
    if (size < 1) return 0;

    // Call your target function with fuzzer-provided data
    my_target_function(data, size);

    return 0;
}
```

Compile and run:

```
clang++ -fsanitize=fuzzer,address -g -O2 harness.cc target.cc -o fuzz
mkdir corpus/
./fuzz corpus/
```

Installation

Prerequisites

- LLVM/Clang compiler (includes libFuzzer)
- LLVM tools for coverage analysis (optional)

Linux (Ubuntu/Debian)

```
apt install clang llvm
```

For the latest LLVM version:

```
# Add LLVM repository from apt.llvm.org
# Then install specific version, e.g.:
apt install clang-18 llvm-18
```

macOS

```
# Using Homebrew
brew install llvm

# Or using Nix
nix-env -i clang
```

Windows

Install Clang through Visual Studio. Refer to Microsoft's documentation for setup instructions.

Recommendation: If possible, fuzz on a local x86_64 VM or rent one on DigitalOcean, AWS, or Hetzner. Linux provides the best support for libFuzzer.

Verification

```
clang++ --version
# Should show LLVM version information
```

Writing a Harness

Harness Structure

The harness is the entry point for the fuzzer. libFuzzer calls the `LLVMFuzzerTestOneInput` function repeatedly with different inputs.

```
#include <stdint.h>
#include <stddef.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // 1. Optional: Validate input size
    if (size < MIN_REQUIRED_SIZE) {
        return 0;  // Reject inputs that are too small
    }

    // 2. Optional: Convert raw bytes to structured data
    // Example: Parse two integers from byte array
    if (size >= 2 * sizeof(uint32_t)) {
        uint32_t a = *(uint32_t*)(data);
        uint32_t b = *(uint32_t*)(data + sizeof(uint32_t));
        my_function(a, b);
    }

    // 3. Call target function
    target_function(data, size);

    // 4. Always return 0 (non-zero reserved for future use)
    return 0;
}
```

Harness Rules

Do	Don't
Handle all input types (empty, huge, malformed)	Call `exit()` - stops fuzzing process
Join all threads before returning	Leave threads running
Keep harness fast and simple	Add excessive logging or complexity
Maintain determinism	Use random number generators or read `/dev/random`
Reset global state between runs	Rely on state from previous executions
Use narrow, focused targets	Mix unrelated data formats (PNG + TCP) in one harness

Rationale:

- **Speed matters:** Aim for 100s-1000s executions per second per core
- **Reproducibility:** Crashes must be reproducible after fuzzing completes
- **Isolation:** Each execution should be independent

Using FuzzedDataProvider for Complex Inputs

For complex inputs (strings, multiple parameters), use the `FuzzedDataProvider` helper:

```
#include <stdint.h>
#include <stddef.h>
#include "FuzzedDataProvider.h"  // From LLVM project

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    FuzzedDataProvider fuzzed_data(data, size);

    // Extract structured data
    size_t allocation_size = fuzzed_data.ConsumeIntegral<size_t>();
    std::vector<char> str1 = fuzzed_data.ConsumeBytesWithTerminator<char>(32, 0xFF);
    std::vector<char> str2 = fuzzed_data.ConsumeBytesWithTerminator<char>(32, 0xFF);

    // Call target with extracted data
    char* result = concat(&str1[0], str1.size(), &str2[0], str2.size(), allocation_size);
    if (result != NULL) {
        free(result);
    }

    return 0;
}
```

Download `FuzzedDataProvider.h` from the LLVM repository.

Interleaved Fuzzing

Use a single harness to test multiple related functions:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size < 1 + 2 * sizeof(int32_t)) {
        return 0;
    }
```

```
    uint8_t mode = data[0];
    int32_t numbers[2];
    memcpy(numbers, data + 1, 2 * sizeof(int32_t));

    // Select function based on first byte
    switch (mode % 4) {
        case 0: add(numbers[0], numbers[1]); break;
        case 1: subtract(numbers[0], numbers[1]); break;
        case 2: multiply(numbers[0], numbers[1]); break;
        case 3: divide(numbers[0], numbers[1]); break;
    }

    return 0;
}
```

> **See Also:** For detailed harness writing techniques, patterns for handling complex inputs, structure-aware fuzzing, and protobuf-based fuzzing, see the **fuzz-harness-writing** technique skill.

Compilation

Basic Compilation

The key flag is `-fsanitize=fuzzer`, which:

- Links the libFuzzer runtime (provides `main` function)
- Enables SanitizerCoverage instrumentation for coverage tracking
- Disables built-in functions like `memcmp`

```
clang++ -fsanitize=fuzzer -g -O2 harness.cc target.cc -o fuzz
```

Flags explained:

- `-fsanitize=fuzzer`: Enable libFuzzer
- `-g`: Add debug symbols (helpful for crash analysis)
- `-O2`: Production-level optimizations (recommended for fuzzing)
- `-DNO_MAIN`: Define macro if your code has a `main` function

With Sanitizers

AddressSanitizer (recommended):

```
clang++ -fsanitize=fuzzer,address -g -O2 -U_FORTIFY_SOURCE harness.cc target.cc -o fuzz
```

Multiple sanitizers:

```
clang++ -fsanitize=fuzzer,address,undefined -g -O2 harness.cc target.cc -o fuzz
```

> **See Also:** For detailed sanitizer configuration, common issues, ASAN_OPTIONS flags, and advanced sanitizer usage, see the **address-sanitizer** and **undefined-behavior-sanitizer** technique skills.

Build Flags

Flag	Purpose
`-fsanitize=fuzzer`	Enable libFuzzer runtime and instrumentation
`-fsanitize=address`	Enable AddressSanitizer (memory error detection)
`-fsanitize=undefined`	Enable UndefinedBehaviorSanitizer
`-fsanitize=fuzzer-no-link`	Instrument without linking fuzzer (for libraries)
`-g`	Include debug symbols
`-O2`	Production optimization level
`-U_FORTIFY_SOURCE`	Disable fortification (can interfere with ASan)

Building Static Libraries

For projects that produce static libraries:

1. Build the library with fuzzing instrumentation:

```
export CC=clang CFLAGS="-fsanitize=fuzzer-no-link -fsanitize=address"
export CXX=clang++ CXXFLAGS="$CFLAGS"
./configure --enable-shared=no
make
```

2. Link the static library with your harness:

```
clang++ -fsanitize=fuzzer -fsanitize=address harness.cc libmylib.a -o fuzz
```

CMake Integration

```
project(FuzzTarget)
cmake_minimum_required(VERSION 3.0)

add_executable(fuzz main.cc harness.cc)
target_compile_definitions(fuzz PRIVATE NO_MAIN=1)
target_compile_options(fuzz PRIVATE -g -O2 -fsanitize=fuzzer -fsanitize=address)
target_link_libraries(fuzz -fsanitize=fuzzer -fsanitize=address)
```

Build with:

```
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ .
cmake --build .
```

Corpus Management

Creating Initial Corpus

Create a directory for the corpus (can start empty):

```
mkdir corpus/
```

Optional but recommended: Provide seed inputs (valid example files):

```
# For a PNG parser:
cp examples/*.png corpus/

# For a protocol parser:
cp test_packets/*.bin corpus/
```

Benefits of seed inputs:

- Fuzzer doesn't start from scratch
- Reaches valid code paths faster
- Significantly improves effectiveness

Corpus Structure

The corpus directory contains:

- Input files that trigger unique code paths
- Minimized versions (libFuzzer automatically minimizes)
- Named by content hash (e.g., `a9993e364706816aba3e25717850c26c9cd0d89d`)

Corpus Minimization

libFuzzer automatically minimizes corpus entries during fuzzing. To explicitly minimize:

```
mkdir minimized_corpus/
./fuzz -merge=1 minimized_corpus/ corpus/
```

This creates a deduplicated, minimized corpus in `minimized_corpus/`.

> **See Also:** For corpus creation strategies, seed selection, format-specific corpus building, and corpus maintenance, see the **fuzzing-corpus** technique skill.

Running Campaigns

Basic Run

```
./fuzz corpus/
```

This runs until a crash is found or you stop it (Ctrl+C).

Recommended: Continue After Crashes

```
./fuzz -fork=1 -ignore_crashes=1 corpus/
```

The `-fork` and `-ignore_crashes` flags (experimental but widely used) allow fuzzing to continue after finding crashes.

Common Options

Control input size:

```
./fuzz -max_len=4000 corpus/
```

Rule of thumb: 2x the size of minimal realistic input.

Set timeout:

```
./fuzz -timeout=2 corpus/
```

Abort test cases that run longer than 2 seconds.

Use a dictionary:

```
./fuzz -dict=./format.dict corpus/
```

Close stdout/stderr (speed up fuzzing):

```
./fuzz -close_fd_mask=3 corpus/
```

See all options:

```
./fuzz -help=1
```

Multi-Core Fuzzing

Option 1: Jobs and workers (recommended):

```
./fuzz -jobs=4 -workers=4 -fork=1 -ignore_crashes=1 corpus/
```

- `-jobs=4`: Run 4 sequential campaigns
- `-workers=4`: Process jobs in parallel with 4 processes
- Test cases are shared between jobs

Option 2: Fork mode:

```
./fuzz -fork=4 -ignore_crashes=1 corpus/
```

Note: For serious multi-core fuzzing, consider switching to AFL++, Honggfuzz, or LibAFL.

Re-executing Test Cases

Re-run a single crash:

```
./fuzz ./crash-a9993e364706816aba3e25717850c26c9cd0d89d
```

Test all inputs in a directory without fuzzing:

```
./fuzz -runs=0 corpus/
```

Interpreting Output

When fuzzing runs, you'll see statistics like:

```
INFO: Seed: 3517090860
INFO: Loaded 1 modules (9 inline 8-bit counters)
#2      INITED cov: 3 ft: 4 corp: 1/1b exec/s: 0 rss: 26Mb
#57     NEW    cov: 4 ft: 5 corp: 2/4b lim: 4 exec/s: 0 rss: 26Mb
```

Output	Meaning
`INITED`	Fuzzing initialized
`NEW`	New coverage found, added to corpus
`REDUCE`	Input minimized while keeping coverage
`cov: N`	Number of coverage edges hit
`corp: X/Yb`	Corpus size: X entries, Y total bytes
`exec/s: N`	Executions per second
`rss: NMb`	Resident memory usage

On crash:

```
==11672== ERROR: libFuzzer: deadly signal
artifact_prefix='./'; Test unit written to ./crash-a9993e364706816aba3e25717850c26c9cd0d89d
0x61,0x62,0x63,
abc
Base64: YWJj
```

The crash is saved to `./crash-<hash>` with the input shown in hex, UTF-8, and Base64.

Reproducibility: Use `-seed=<value>` to reproduce a fuzzing campaign (single-core only).

Fuzzing Dictionary

Dictionaries help the fuzzer discover interesting inputs faster by providing hints about the input format.

Dictionary Format

Create a text file with quoted strings (one per line):

```
# Lines starting with '#' are comments

# Magic bytes
magic="\x89PNG"
magic2="IEND"

# Keywords
"GET"
"POST"
"Content-Type"

# Hex sequences
delimiter="\xFF\xD8\xFF"
```

Using a Dictionary

```
./fuzz -dict=./format.dict corpus/
```

Generating a Dictionary

From header files:

```
grep -o '".*"' header.h > header.dict
```

From man pages:

```
man curl | grep -oP '^\s*(--|-)\K\S+' | sed 's/[,.]$//' | sed 's/^/"&/; s/$/&"/' | sort -u > man.dict
```

From binary strings:

```
strings ./binary | sed 's/^/"&/; s/$/&"/' > strings.dict
```

Using LLMs: Ask ChatGPT or similar to generate a dictionary for your format (e.g., "Generate a libFuzzer dictionary for a JSON parser").

> **See Also:** For advanced dictionary generation, format-specific dictionaries, and dictionary optimization strategies, see the **fuzzing-dictionaries** technique skill.

Coverage Analysis

While libFuzzer shows basic coverage stats (`cov: N`), detailed coverage analysis requires additional tools.

Source-Based Coverage

1. Recompile with coverage instrumentation:

```
clang++ -fsanitize=fuzzer -fprofile-instr-generate -fcoverage-mapping harness.cc target.cc -o fuzz
```

2. Run fuzzer to collect coverage:

```
LLVM_PROFILE_FILE="coverage-%p.profraw" ./fuzz -runs=10000 corpus/
```

3. Merge coverage data:

```
llvm-profdata merge -sparse coverage-*.profraw -o coverage.profdata
```

4. Generate coverage report:

```
llvm-cov show ./fuzz -instr-profile=coverage.profdata
```

5. Generate HTML report:

```
llvm-cov show ./fuzz -instr-profile=coverage.profdata -format=html > coverage.html
```

Improving Coverage

Tips:

- Provide better seed inputs in corpus
- Use dictionaries for format-aware fuzzing
- Check if harness properly exercises target
- Consider structure-aware fuzzing for complex formats
- Run longer campaigns (days/weeks)

> **See Also:** For detailed coverage analysis techniques, identifying coverage gaps, systematic coverage improvement, and comparing coverage across fuzzers, see the **coverage-analysis** technique skill.

Sanitizer Integration

AddressSanitizer (ASan)

ASan detects memory errors like buffer overflows and use-after-free bugs. **Highly recommended for fuzzing.**

Enable ASan:

```
clang++ -fsanitize=fuzzer,address -g -O2 -U_FORTIFY_SOURCE harness.cc target.cc -o fuzz
```

Example ASan output:

```
==1276163==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020000c4ab1
WRITE of size 1 at 0x6020000c4ab1 thread T0
    #0 0x55555568631a in check_buf(char*, unsigned long) main.cc:13:25
    #1 0x5555556860bf in LLVMFuzzerTestOneInput harness.cc:7:3
```

Configure ASan with environment variables:

```
ASAN_OPTIONS=verbosity=1:abort_on_error=1 ./fuzz corpus/
```

Important flags:

- `verbosity=1`: Show ASan is active
- `detect_leaks=0`: Disable leak detection (leaks reported at end)
- `abort_on_error=1`: Call `abort()` instead of `_exit()` on errors

Drawbacks:

- 2-4x slowdown
- Requires ~20TB virtual memory (disable memory limits: `-rss_limit_mb=0`)
- Best supported on Linux

> **See Also:** For comprehensive ASan configuration, common pitfalls, symbolization, and combining with other sanitizers, see the **address-sanitizer** technique skill.

UndefinedBehaviorSanitizer (UBSan)

UBSan detects undefined behavior like integer overflow, null pointer dereference, etc.

Enable UBSan:

```
clang++ -fsanitize=fuzzer,undefined -g -O2 harness.cc target.cc -o fuzz
```

Combine with ASan:

```
clang++ -fsanitize=fuzzer,address,undefined -g -O2 harness.cc target.cc -o fuzz
```

MemorySanitizer (MSan)

MSan detects uninitialized memory reads. More complex to use (requires rebuilding all dependencies).

```
clang++ -fsanitize=fuzzer,memory -g -O2 harness.cc target.cc -o fuzz
```

Common Sanitizer Issues

Issue	Solution
ASan slows fuzzing too much	Use `-fsanitize-recover=address` for non-fatal errors
Out of memory	Set `ASAN_OPTIONS=rss_limit_mb=0` or `-rss_limit_mb=0`
Stack exhaustion	Increase stack size: `ASAN_OPTIONS=stack_size=8388608`
False positives with `_FORTIFY_SOURCE`	Use `-U_FORTIFY_SOURCE` flag
MSan reports in dependencies	Rebuild all dependencies with `-fsanitize=memory`

Real-World Examples

Example 1: Fuzzing libpng

libpng is a widely-used library for reading/writing PNG images. Bugs can lead to security issues.

1. Get source code:

```
curl -L -O https://downloads.sourceforge.net/project/libpng/libpng16/1.6.37/libpng-1.6.37.tar.xz
tar xf libpng-1.6.37.tar.xz
cd libpng-1.6.37/
```

2. Install dependencies:

```
apt install zlib1g-dev
```

3. Compile with fuzzing instrumentation:

```
export CC=clang CFLAGS="-fsanitize=fuzzer-no-link -fsanitize=address"
export CXX=clang++ CXXFLAGS="$CFLAGS"
./configure --enable-shared=no
make
```

4. Get a harness (or write your own):

```
curl -O https://raw.githubusercontent.com/glennrp/libpng/f8e5fa92b0e37ab597616f554bee254157998227/contrib/oss-fuzz/libpng_read_fuzzer.cc
```

5. Prepare corpus and dictionary:

```
mkdir corpus/
curl -o corpus/input.png https://raw.githubusercontent.com/glennrp/libpng/acfd50ae0ba3198ad734e5d4dec2b05341e50924/contrib/pngsuite/iftp1n3p08.png
curl -O https://raw.githubusercontent.com/glennrp/libpng/2fff013a6935967960a5ae626fc21432807933dd/contrib/oss-fuzz/png.dict
```

6. Link and compile fuzzer:

```
clang++ -fsanitize=fuzzer -fsanitize=address libpng_read_fuzzer.cc .libs/libpng16.a -lz -o
fuzz
```

7. Run fuzzing campaign:

```
./fuzz -close_fd_mask=3 -dict=./png.dict corpus/
```

Example 2: Simple Division Bug

Harness that finds a division-by-zero bug:

```
#include <stdint.h>
#include <stddef.h>

double divide(uint32_t numerator, uint32_t denominator) {
    // Bug: No check if denominator is zero
    return numerator / denominator;
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if(size != 2 * sizeof(uint32_t)) {
        return 0;
    }

    uint32_t numerator = *(uint32_t*)(data);
    uint32_t denominator = *(uint32_t*)(data + sizeof(uint32_t));

    divide(numerator, denominator);

    return 0;
}
```

Compile and fuzz:

```
clang++ -fsanitize=fuzzer harness.cc -o fuzz
./fuzz
```

The fuzzer will quickly find inputs causing a crash.

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Start with single-core, switch to AFL++ for multi-core	libFuzzer harnesses work with AFL++
Use dictionaries for structured formats	10-100x faster bug discovery
Close file descriptors with `-close_fd_mask=3`	Speed boost if SUT writes output
Set reasonable `-max_len`	Prevents wasted time on huge inputs
Run for days/weeks, not minutes	Coverage plateaus take time to break
Use seed corpus from test suites	Starts fuzzing from valid inputs

Structure-Aware Fuzzing

For highly structured inputs (e.g., complex protocols, file formats), use libprotobuf-mutator:

- Define input structure using Protocol Buffers
- libFuzzer mutates protobuf messages (structure-preserving mutations)
- Harness converts protobuf to native format

See structure-aware fuzzing documentation for details.

Custom Mutators

libFuzzer allows custom mutators for specialized fuzzing:

```
extern "C" size_t LLVMFuzzerCustomMutator(uint8_t *Data, size_t Size,
                                          size_t MaxSize, unsigned int Seed) {
    // Custom mutation logic
    return new_size;
}

extern "C" size_t LLVMFuzzerCustomCrossOver(const uint8_t *Data1, size_t Size1,
                                            const uint8_t *Data2, size_t Size2,
                                            uint8_t *Out, size_t MaxOutSize,
                                            unsigned int Seed) {
    // Custom crossover logic
```

```
    return new_size;
}
```

Performance Tuning

Setting	Impact
`-close_fd_mask=3`	Closes stdout/stderr, speeds up fuzzing
`-max_len=<reasonable_size>`	Avoids wasting time on huge inputs
`-timeout=<seconds>`	Detects hangs, prevents stuck executions
Disable ASan for baseline	2-4x speed boost (but misses memory bugs)
Use `-jobs` and `-workers`	Limited multi-core support
Run on Linux	Best platform support and performance

Troubleshooting

Problem	Cause	Solution
No crashes found after hours	Poor corpus, low coverage	Add seed inputs, use dictionary, check harness
Very slow executions/sec (<100)	Target too complex, excessive logging	Optimize target, use `-close_fd_mask=3`, reduce logging
Out of memory	ASan's 20TB virtual memory	Set `-rss_limit_mb=0` to disable RSS limit
Fuzzer stops after first crash	Default behavior	Use `-fork=1 -ignore_crashes=1` to continue
Can't reproduce crash	Non-determinism in harness/target	Remove random number generation, global state
Linking errors with `-fsanitize=fuzzer`	Missing libFuzzer runtime	Ensure using Clang, check LLVM installation
GCC project won't compile with Clang	GCC-specific code	Switch to AFL++ with `gcc_plugin` instead
Coverage not improving	Corpus plateau	Run longer, add dictionary, improve seeds, check coverage report
Crashes but ASan doesn't trigger	Memory error not detected without ASan	Recompile with `-fsanitize=address`

Related Skills

Technique Skills

Skill	Use Case
fuzz-harness-writing	Detailed guidance on writing effective harnesses, structure-aware fuzzing, and FuzzedDataProvider usage
address-sanitizer	Memory error detection configuration, ASAN_OPTIONS, and troubleshooting
undefined-behavior-sanitizer	Detecting undefined behavior during fuzzing
coverage-analysis	Measuring fuzzing effectiveness and identifying untested code paths
fuzzing-corpus	Building and managing seed corpora, corpus minimization strategies
fuzzing-dictionaries	Creating format-specific dictionaries for faster bug discovery

Related Fuzzers

Skill	When to Consider
aflpp	When you need serious multi-core fuzzing, or when libFuzzer coverage plateaus
honggfuzz	When you want hardware-based coverage feedback on Linux
libafl	When building custom fuzzers or conducting fuzzing research

Resources

Official Documentation

- LLVM libFuzzer Documentation - Official reference
- libFuzzer Tutorial by Google - Step-by-step guide
- SanitizerCoverage - Coverage instrumentation details

Advanced Topics

- Structure-Aware Fuzzing with libprotobuf-mutator

- Split Inputs in libFuzzer
- FuzzedDataProvider Header

Example Projects

- OSS-Fuzz - Continuous fuzzing for open-source projects (many libFuzzer examples)
- AFL++ Dictionary Collection - Reusable dictionaries

Revision #6
Created 2026-02-18 08:40:12 UTC by John
Updated 2026-07-26 20:01:32 UTC by John