/harness-writing

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

---

name: harness-writing type: technique description: > Techniques for writing effective fuzzing harnesses across languages. Use when creating new fuzz targets or improving existing harness code.

Writing Fuzzing Harnesses

A fuzzing harness is the entrypoint function that receives random data from the fuzzer and routes it to your system under test (SUT). The quality of your harness directly determines which code paths get exercised and whether critical bugs are found. A poorly written harness can miss entire subsystems or produce non-reproducible crashes.

Overview

The harness is the bridge between the fuzzer's random byte generation and your application's API. It must parse raw bytes into meaningful inputs, call target functions, and handle edge cases gracefully. The most important part of any fuzzing setup is the harness—if written poorly, critical parts of your application may not be covered.

Key Concepts

Concept	Description
Harness	Function that receives fuzzer input and calls target code under test
SUT	System Under Test—the code being fuzzed
Entry point	Function signature required by the fuzzer (e.g., `LLVMFuzzerTestOneInput`)
FuzzedDataProvider	Helper class for structured extraction of typed data from raw bytes
Determinism	Property that ensures same input always produces same behavior
Interleaved fuzzing	Single harness that exercises multiple operations based on input

When to Apply

Apply this technique when:

- Creating a new fuzz target for the first time
- Fuzz campaign has low code coverage or isn't finding bugs
- Crashes found during fuzzing are not reproducible
- Target API requires complex or structured inputs
- Multiple related functions should be tested together

Skip this technique when:

- Using existing well-tested harnesses from your project
- Tool provides automatic harness generation that meets your needs
- Target already has comprehensive fuzzing infrastructure

Quick Reference

Task	Pattern

Minimal C++ harness	`extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)`
Minimal Rust harness	`fuzz_target!(
Size validation	`if (size < MIN_SIZE) return 0;`
Cast to integers	`uint32_t val = *(uint32_t*)(data);`
Use FuzzedDataProvider	`FuzzedDataProvider fuzzed_data(data, size);`
Extract typed data (C++)	`auto val = fuzzed_data.ConsumeIntegral<uint32_t>();`
Extract string (C++)	`auto str = fuzzed_data.ConsumeBytesWithTerminator<char>(32, 0xFF);`

Step-by-Step

Step 1: Identify Entry Points

Find functions in your codebase that:

- Accept external input (parsers, validators, protocol handlers)
- Parse complex data formats (JSON, XML, binary protocols)
- Perform security-critical operations (authentication, cryptography)
- Have high cyclomatic complexity or many branches

Good targets are typically:

- Protocol parsers
- File format parsers
- Serialization/deserialization functions
- Input validation routines

Step 2: Write Minimal Harness

Start with the simplest possible harness that calls your target function:

C/C++:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    target_function(data, size);
    return 0;
}
```

Rust:

```
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: &[u8]| {
    target_function(data);
});
```

Step 3: Add Input Validation

Reject inputs that are too small or too large to be meaningful:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Ensure minimum size for meaningful input
    if (size < MIN_INPUT_SIZE || size > MAX_INPUT_SIZE) {
        return 0;
    }
    target_function(data, size);
    return 0;
}
```

Rationale: The fuzzer generates random inputs of all sizes. Your harness must handle empty, tiny, huge, or malformed inputs without causing unexpected issues in the harness itself (crashes in the SUT are fine—that's what we're looking for).

Step 4: Structure the Input

For APIs that require typed data (integers, strings, etc.), use casting or helpers like `FuzzedDataProvider`:

Simple casting:

```
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;
}
```

Using FuzzedDataProvider:

```
#include "FuzzedDataProvider.h"

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

    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);

    concat(&str1[0], str1.size(), &str2[0], str2.size(), allocation_size);
    return 0;
}
```

Step 5: Test and Iterate

Run the fuzzer and monitor:

- Code coverage (are all interesting paths reached?)
- Executions per second (is it fast enough?)
- Crash reproducibility (can you reproduce crashes with saved inputs?)

Iterate on the harness to improve these metrics.

Common Patterns

Pattern: Beyond Byte Arrays—Casting to Integers

Use Case: When target expects primitive types like integers or floats

Implementation:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Ensure exactly 2 4-byte numbers
```

```
    if (size != 2 * sizeof(uint32_t)) {
        return 0;
    }

    // Split input into two integers
    uint32_t numerator = *(uint32_t*)(data);
    uint32_t denominator = *(uint32_t*)(data + sizeof(uint32_t));

    divide(numerator, denominator);
    return 0;
}
```

Rust equivalent:

```
fuzz_target!(|data: &[u8]| {
    if data.len() != 2 * std::mem::size_of::<i32>() {
        return;
    }

    let numerator = i32::from_ne_bytes([data[0], data[1], data[2], data[3]]);
    let denominator = i32::from_ne_bytes([data[4], data[5], data[6], data[7]]);

    divide(numerator, denominator);
});
```

Why it works: Any 8-byte input is valid. The fuzzer learns that inputs must be exactly 8 bytes, and every bit flip produces a new, potentially interesting input.

Pattern: FuzzedDataProvider for Complex Inputs

Use Case: When target requires multiple strings, integers, or variable-length data

Implementation:

```
#include "FuzzedDataProvider.h"

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

```
    // Extract different types of data
    size_t allocation_size = fuzzed_data.ConsumeIntegral<size_t>();

    // Consume variable-length strings with terminator
    std::vector<char> str1 = fuzzed_data.ConsumeBytesWithTerminator<char>(32, 0xFF);
    std::vector<char> str2 = fuzzed_data.ConsumeBytesWithTerminator<char>(32, 0xFF);

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

    return 0;
}
```

Why it helps: `FuzzedDataProvider` handles the complexity of extracting structured data from a byte stream. It's particularly useful for APIs that need multiple parameters of different types.

Pattern: Interleaved Fuzzing

Use Case: When multiple related operations should be tested in a single harness

Implementation:

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

    // First byte selects operation
    uint8_t mode = data[0];

    // Next bytes are operands
    int32_t numbers[2];
    memcpy(numbers, data + 1, 2 * sizeof(int32_t));

    int32_t result = 0;
    switch (mode % 4) {
        case 0:
            result = add(numbers[0], numbers[1]);
```

```
            break;
        case 1:
            result = subtract(numbers[0], numbers[1]);
            break;
        case 2:
            result = multiply(numbers[0], numbers[1]);
            break;
        case 3:
            result = divide(numbers[0], numbers[1]);
            break;
    }

    // Prevent compiler from optimizing away the calls
    printf("%d", result);
    return 0;
}
```

Advantages:

- Faster to write one harness than multiple individual harnesses
- Single shared corpus means interesting inputs for one operation may be interesting for others
- Can discover bugs in interactions between operations

When to use:

- Operations share similar input types
- Operations are logically related (e.g., arithmetic operations, CRUD operations)
- Single corpus makes sense across all operations

Pattern: Structure-Aware Fuzzing with Arbitrary (Rust)

Use Case: When fuzzing Rust code that uses custom structs

Implementation:

```
use arbitrary::Arbitrary;

#[derive(Debug, Arbitrary)]
pub struct Name {
```

```
    data: String
}

impl Name {
    pub fn check_buf(&self) {
        let data = self.data.as_bytes();
        if data.len() > 0 && data[0] == b'a' {
            if data.len() > 1 && data[1] == b'b' {
                if data.len() > 2 && data[2] == b'c' {
                    process::abort();
                }
            }
        }
    }
}
```

Harness with arbitrary:

```
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: your_project::Name| {
    data.check_buf();
});
```

Add to Cargo.toml:

```
[dependencies]
arbitrary = { version = "1", features = ["derive"] }
```

Why it helps: The `arbitrary` crate automatically handles deserialization of raw bytes into your Rust structs, reducing boilerplate and ensuring valid struct construction.

Limitation: The arbitrary crate doesn't offer reverse serialization, so you can't manually construct byte arrays that map to specific structs. This works best when starting from an empty corpus (fine for libFuzzer, problematic for AFL++).

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Start with parsers	High bug density, clear entry points, easy to harness
Mock I/O operations	Prevents hangs from blocking I/O, enables determinism
Use FuzzedDataProvider	Simplifies extraction of structured data from raw bytes
Reset global state	Ensures each iteration is independent and reproducible
Free resources in harness	Prevents memory exhaustion during long campaigns
Avoid logging in harness	Logging is slow—fuzzing needs 100s-1000s exec/sec
Test harness manually first	Run harness with known inputs before starting campaign
Check coverage early	Ensure harness reaches expected code paths

Structure-Aware Fuzzing with Protocol Buffers

For highly structured input formats, consider using Protocol Buffers as an intermediate format with custom mutators:

```
// Define your input format in .proto file
// Use libprotobuf-mutator to generate valid mutations
// This ensures fuzzer mutates message contents, not the protobuf encoding itself
```

This approach is more setup but prevents the fuzzer from wasting time on unparseable inputs. See structure-aware fuzzing documentation for details.

Handling Non-Determinism

Problem: Random values or timing dependencies cause non-reproducible crashes.

Solutions:

- Replace `rand()` with deterministic PRNG seeded from fuzzer input:

  ```
  uint32_t seed = fuzzed_data.ConsumeIntegral<uint32_t>();
  srand(seed);
  ```

- Mock system calls that return time, PIDs, or random data
- Avoid reading from `/dev/random` or `/dev/urandom`

Resetting Global State

If your SUT uses global state (singletons, static variables), reset it between iterations:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Reset global state before each iteration
    global_reset();

    target_function(data, size);

    // Clean up resources
    global_cleanup();
    return 0;
}
```

Rationale: Global state can cause crashes after N iterations rather than on a specific input, making bugs non-reproducible.

Practical Harness Rules

Follow these rules to ensure effective fuzzing harnesses:

Rule	Rationale
Handle all input sizes	Fuzzer generates empty, tiny, huge inputs—harness must handle gracefully
Never call `exit()`	Calling `exit()` stops the fuzzer process. Use `abort()` in SUT if needed
Join all threads	Each iteration must run to completion before next iteration starts
Be fast	Aim for 100s-1000s executions/sec. Avoid logging, high complexity, excess memory
Maintain determinism	Same input must always produce same behavior for reproducibility
Avoid global state	Global state reduces reproducibility—reset between iterations if unavoidable
Use narrow targets	Don't fuzz PNG and TCP in same harness—different formats need separate targets
Free resources	Prevent memory leaks that cause resource exhaustion during long campaigns

Note: These guidelines apply not just to harness code, but to the entire SUT. If the SUT violates these rules, consider patching it (see the fuzzing obstacles technique).

Anti-Patterns

Anti-Pattern	Problem	Correct Approach
Global state without reset	Non-deterministic crashes	Reset all globals at start of harness
Blocking I/O or network calls	Hangs fuzzer, wastes time	Mock I/O, use in-memory buffers
Memory leaks in harness	Resource exhaustion kills campaign	Free all allocations before returning
Calling `exit()` in SUT	Stops entire fuzzing process	Use `abort()` or return error codes
Heavy logging in harness	Reduces exec/sec by orders of magnitude	Disable logging during fuzzing
Too many operations per iteration	Slows down fuzzer	Keep iterations fast and focused
Mixing unrelated input formats	Corpus entries not useful across formats	Separate harnesses for different formats
Not validating input size	Harness crashes on edge cases	Check `size` before accessing `data`

Tool-Specific Guidance

libFuzzer

Harness signature:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Your code here
    return 0;  // Non-zero return is reserved for future use
}
```

Compilation:

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

Integration tips:

- Use `FuzzedDataProvider.h` for structured input extraction
- Compile with `-fsanitize=fuzzer` to link the fuzzing runtime
- Add sanitizers (`-fsanitize=address,undefined`) to detect more bugs
- Use `-g` for better stack traces when crashes occur
- libFuzzer can start with empty corpus—no seed inputs required

Running:

```
./fuzz_target corpus_dir/
```

Resources:

- FuzzedDataProvider header
- libFuzzer documentation

AFL++

AFL++ supports multiple harness styles. For best performance, use persistent mode:

Persistent mode harness:

```
#include <unistd.h>

int main(int argc, char **argv) {
    #ifdef __AFL_HAVE_MANUAL_CONTROL
        __AFL_INIT();
    #endif

    unsigned char buf[MAX_SIZE];

    while (__AFL_LOOP(10000)) {
        // Read input from stdin
        ssize_t len = read(0, buf, sizeof(buf));
        if (len <= 0) break;

        // Call target function
        target_function(buf, len);
    }

    return 0;
}
```

Compilation:

```
afl-clang-fast++ -g harness.cc -o fuzz_target
```

Integration tips:

- Use persistent mode (`__AFL_LOOP`) for 10-100x speedup

- Consider deferred initialization (`__AFL_INIT()`) to skip setup overhead
- AFL++ requires at least one seed input in the corpus directory
- Use `AFL_USE_ASAN=1` or `AFL_USE_UBSAN=1` for sanitizer builds

Running:

```
afl-fuzz -i seeds/ -o findings/ -- ./fuzz_target
```

cargo-fuzz (Rust)

Harness signature:

```
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: &[u8]| {
    // Your code here
});
```

With structured input (arbitrary crate):

```
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: YourStruct| {
    data.check();
});
```

Creating harness:

```
cargo fuzz init
cargo fuzz add my_target
```

Integration tips:

- Use `arbitrary` crate for automatic struct deserialization
- cargo-fuzz wraps libFuzzer, so all libFuzzer features work
- Compile with sanitizers automatically via cargo-fuzz
- Harnesses go in `fuzz/fuzz_targets/` directory

Running:

```
cargo +nightly fuzz run my_target
```

Resources:

- cargo-fuzz documentation
- arbitrary crate

go-fuzz

Harness signature:

```
// +build gofuzz

package mypackage

func Fuzz(data []byte) int {
    // Call target function
    target(data)

    // Return codes:
    // -1 if input is invalid
    //  0 if input is valid but not interesting
    //  1 if input is interesting (e.g., added new coverage)
    return 0
}
```

Building:

```
go-fuzz-build
```

Integration tips:

- Return 1 for inputs that add coverage (optional—fuzzer can detect automatically)
- Return -1 for invalid inputs to deprioritize similar mutations
- go-fuzz handles persistence automatically

Running:

```
go-fuzz -bin=./mypackage-fuzz.zip -workdir=fuzz
```

Troubleshooting

Issue	Cause	Solution
Low executions/sec	Harness is too slow (logging, I/O, complexity)	Profile harness, remove bottlenecks, mock I/O
No crashes found	Coverage not reaching buggy code	Check coverage, improve harness to reach more paths
Non-reproducible crashes	Non-determinism or global state	Remove randomness, reset globals between iterations
Fuzzer exits immediately	Harness calls `exit()`	Replace `exit()` with `abort()` or return error
Out of memory errors	Memory leaks in harness or SUT	Free allocations, use leak sanitizer to find leaks
Crashes on empty input	Harness doesn't validate size	Add `if (size < MIN_SIZE) return 0;`
Corpus not growing	Inputs too constrained or format too strict	Use FuzzedDataProvider or structure-aware fuzzing

Related Skills

Tools That Use This Technique

Skill	How It Applies
libfuzzer	Uses `LLVMFuzzerTestOneInput` harness signature with FuzzedDataProvider
aflpp	Supports persistent mode harnesses with `__AFL_LOOP` for performance
cargo-fuzz	Uses Rust-specific `fuzz_target!` macro with arbitrary crate integration
atheris	Python harness takes bytes, calls Python functions
ossfuzz	Requires harnesses in specific directory structure for cloud fuzzing

Related Techniques

Skill	Relationship

coverage-analysis	Measure harness effectiveness—are you reaching target code?
address-sanitizer	Detects bugs found by harness (buffer overflows, use-after-free)
fuzzing-dictionary	Provide tokens to help fuzzer pass format checks in harness
fuzzing-obstacles	Patch SUT when it violates harness rules (exit, non-determinism)

Resources

Key External Resources

Split Inputs in libFuzzer - Google Fuzzing Docs Explains techniques for handling multiple input parameters in a single fuzzing harness, including use of magic separators and FuzzedDataProvider.

Structure-Aware Fuzzing with Protocol Buffers Advanced technique using protobuf as intermediate format with custom mutators to ensure fuzzer mutates message contents rather than format encoding.

libFuzzer Documentation Official LLVM documentation covering harness requirements, best practices, and advanced features.

cargo-fuzz Book Comprehensive guide to writing Rust fuzzing harnesses with cargo-fuzz and the arbitrary crate.

Video Resources

- Effective File Format Fuzzing - Conference talk on writing harnesses for file format parsers
- Modern Fuzzing of C/C++ Projects - Tutorial covering harness design patterns

---

Revision #5
Created 2026-02-18 08:40:11 UTC by John
Updated 2026-06-21 20:01:26 UTC by John