/cargo-fuzz

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

---

name: cargo-fuzz type: fuzzer description: > cargo-fuzz is the de facto fuzzing tool for Rust projects using Cargo. Use for fuzzing Rust code with libFuzzer backend.

cargo-fuzz

cargo-fuzz is the de facto choice for fuzzing Rust projects when using Cargo. It uses libFuzzer as the backend and provides a convenient Cargo subcommand that automatically enables relevant compilation flags for your Rust project, including support for sanitizers like AddressSanitizer.

When to Use

cargo-fuzz is currently the primary and most mature fuzzing solution for Rust projects using Cargo.

Fuzzer	Best For	Complexity
cargo-fuzz	Cargo-based Rust projects, quick setup	Low

Fuzzer	Best For	Complexity
AFL++	Multi-core fuzzing, non-Cargo projects	Medium
LibAFL	Custom fuzzers, research, advanced use cases	High

Choose cargo-fuzz when:

- Your project uses Cargo (required)
- You want simple, quick setup with minimal configuration
- You need integrated sanitizer support
- You're fuzzing Rust code with or without unsafe blocks

Quick Start

```
#![no_main]

use libfuzzer_sys::fuzz_target;

fn harness(data: &[u8]) {
    your_project::check_buf(data);
}

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

Initialize and run:

```
cargo fuzz init
# Edit fuzz/fuzz_targets/fuzz_target_1.rs with your harness
cargo +nightly fuzz run fuzz_target_1
```

Installation

cargo-fuzz requires the nightly Rust toolchain because it uses features only available in nightly.

Prerequisites

- Rust and Cargo installed via rustup
- Nightly toolchain

Linux/macOS

```
# Install nightly toolchain
rustup install nightly

# Install cargo-fuzz
cargo install cargo-fuzz
```

Verification

```
cargo +nightly --version
cargo fuzz --version
```

Writing a Harness

Project Structure

cargo-fuzz works best when your code is structured as a library crate. If you have a binary project, split your `main.rs` into:

```
src/main.rs  # Entry point (main function)
src/lib.rs   # Code to fuzz (public functions)
Cargo.toml
```

Initialize fuzzing:

```
cargo fuzz init
```

This creates:

```
fuzz/
├── Cargo.toml
└── fuzz_targets/
    └── fuzz_target_1.rs
```

Harness Structure

```
#![no_main]

use libfuzzer_sys::fuzz_target;

fn harness(data: &[u8]) {
    // 1. Validate input size if needed
    if data.is_empty() {
        return;
    }

    // 2. Call target function with fuzz data
    your_project::target_function(data);
}

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

Harness Rules

Do	Don't
Structure code as library crate	Keep everything in main.rs
Use `fuzz_target!` macro	Write custom main function
Handle `Result::Err` gracefully	Panic on expected errors
Keep harness deterministic	Use random number generators

> **See Also:** For detailed harness writing techniques and structure-aware fuzzing with the `arbitrary` crate, see the **fuzz-harness-writing** technique skill.

Structure-Aware Fuzzing

cargo-fuzz integrates with the arbitrary crate for structure-aware fuzzing:

```
// In your library crate
use arbitrary::Arbitrary;

#[derive(Debug, Arbitrary)]
pub struct Name {
    data: String
}
```

```
// In your fuzz target
#![no_main]
use libfuzzer_sys::fuzz_target;

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

Add to your library's `Cargo.toml`:

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

Running Campaigns

Basic Run

```
cargo +nightly fuzz run fuzz_target_1
```

Without Sanitizers (Safe Rust)

If your project doesn't use unsafe Rust, disable sanitizers for 2x performance boost:

```
cargo +nightly fuzz run --sanitizer none fuzz_target_1
```

Check if your project uses unsafe code:

```
cargo install cargo-geiger
cargo geiger
```

Re-executing Test Cases

```
# Run a specific test case (e.g., a crash)
cargo +nightly fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/crash-<hash>

# Run all corpus entries without fuzzing
cargo +nightly fuzz run fuzz_target_1 fuzz/corpus/fuzz_target_1 -- -runs=0
```

Using Dictionaries

```
cargo +nightly fuzz run fuzz_target_1 -- -dict=./dict.dict
```

Interpreting Output

Output	Meaning
`NEW`	New coverage-increasing input discovered
`pulse`	Periodic status update
`INITED`	Fuzzer initialized successfully
Crash with stack trace	Bug found, saved to `fuzz/artifacts/`

Corpus location: `fuzz/corpus/fuzz_target_1/` Crashes location: `fuzz/artifacts/fuzz_target_1/`

Sanitizer Integration

AddressSanitizer (ASan)

ASan is enabled by default and detects memory errors:

```
cargo +nightly fuzz run fuzz_target_1
```

Disabling Sanitizers

For pure safe Rust (no unsafe blocks in your code or dependencies):

```
cargo +nightly fuzz run --sanitizer none fuzz_target_1
```

Performance impact: ASan adds ~2x overhead. Disable for safe Rust to improve fuzzing speed.

Checking for Unsafe Code

```
cargo install cargo-geiger
cargo geiger
```

> **See Also:** For detailed sanitizer configuration, flags, and troubleshooting, see the **address-sanitizer** technique skill.

Coverage Analysis

cargo-fuzz integrates with Rust's coverage tools to analyze fuzzing effectiveness.

Prerequisites

```
rustup toolchain install nightly --component llvm-tools-preview
cargo install cargo-binutils
cargo install rustfilt
```

Generating Coverage Reports

```
# Generate coverage data from corpus
cargo +nightly fuzz coverage fuzz_target_1
```

Create coverage generation script:

```
cat <<'EOF' > ./generate_html
#!/bin/sh
if [ $# -lt 1 ]; then
    echo "Error: Name of fuzz target is required."
    echo "Usage: $0 fuzz_target [sources...]"
    exit 1
fi
FUZZ_TARGET="$1"
shift
```

```
SRC_FILTER="$@"
TARGET=$(rustc -vV | sed -n 's|host: ||p')
cargo +nightly cov -- show -Xdemangler=rustfilt \
  "target/$TARGET/coverage/$TARGET/release/$FUZZ_TARGET" \
  -instr-profile="fuzz/coverage/$FUZZ_TARGET/coverage.profdata"  \
  -show-line-counts-or-regions -show-instantiations  \
  -format=html -o fuzz_html/ $SRC_FILTER
EOF
chmod +x ./generate_html
```

Generate HTML report:

```
./generate_html fuzz_target_1 src/lib.rs
```

HTML report saved to: `fuzz_html/`

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

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Start with a seed corpus	Dramatically speeds up initial coverage discovery
Use `--sanitizer none` for safe Rust	2x performance improvement
Check coverage regularly	Identifies gaps in harness or seed corpus
Use dictionaries for parsers	Helps overcome magic value checks
Structure code as library	Required for cargo-fuzz integration

libFuzzer Options

Pass options to libFuzzer after `--`:

```
# See all options
cargo +nightly fuzz run fuzz_target_1 -- -help=1

# Set timeout per run
cargo +nightly fuzz run fuzz_target_1 -- -timeout=10

# Use dictionary
cargo +nightly fuzz run fuzz_target_1 -- -dict=dict.dict

# Limit maximum input size
cargo +nightly fuzz run fuzz_target_1 -- -max_len=1024
```

Multi-Core Fuzzing

```
# Experimental forking support (not recommended)
cargo +nightly fuzz run --jobs 1 fuzz_target_1
```

Note: The multi-core fuzzing feature is experimental and not recommended. For parallel fuzzing, consider running multiple instances manually or using AFL++.

Real-World Examples

Example: ogg Crate

The ogg crate parses Ogg media container files. Parsers are excellent fuzzing targets because they handle untrusted data.

```
# Clone and initialize
git clone https://github.com/RustAudio/ogg.git
cd ogg/
cargo fuzz init
```

Harness at `fuzz/fuzz_targets/fuzz_target_1.rs`:

```
#![no_main]

use ogg::{PacketReader, PacketWriter};
use ogg::writing::PacketWriteEndInfo;
```

```
use std::io::Cursor;
use libfuzzer_sys::fuzz_target;

fn harness(data: &[u8]) {
    let mut pck_rdr = PacketReader::new(Cursor::new(data.to_vec()));
    pck_rdr.delete_unread_packets();

    let output = Vec::new();
    let mut pck_wtr = PacketWriter::new(Cursor::new(output));

    if let Ok(_) = pck_rdr.read_packet() {
        if let Ok(r) = pck_rdr.read_packet() {
            match r {
                Some(pck) => {
                    let inf = if pck.last_in_stream() {
                        PacketWriteEndInfo::EndStream
                    } else if pck.last_in_page() {
                        PacketWriteEndInfo::EndPage
                    } else {
                        PacketWriteEndInfo::NormalPacket
                    };
                    let stream_serial = pck.stream_serial();
                    let absgp_page = pck.absgp_page();
                    let _ = pck_wtr.write_packet(
                        pck.data, stream_serial, inf, absgp_page
                    );
                }
                None => return,
            }
        }
    }
}

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

Seed the corpus:

```
mkdir fuzz/corpus/fuzz_target_1/
curl -o fuzz/corpus/fuzz_target_1/320x240.ogg \
  https://commons.wikimedia.org/wiki/File:320x240.ogg
```

Run:

```
cargo +nightly fuzz run fuzz_target_1
```

Analyze coverage:

```
cargo +nightly fuzz coverage fuzz_target_1
./generate_html fuzz_target_1 src/lib.rs
```

Troubleshooting

Problem	Cause	Solution
"requires nightly" error	Using stable toolchain	Use `cargo +nightly fuzz`
Slow fuzzing performance	ASan enabled for safe Rust	Add `--sanitizer none` flag
"cannot find binary"	No library crate	Move code from `main.rs` to `lib.rs`
Sanitizer compilation issues	Wrong nightly version	Try different nightly: `rustup install nightly-2024-01-01`
Low coverage	Missing seed corpus	Add sample inputs to `fuzz/corpus/fuzz_target_1/`
Magic value not found	No dictionary	Create dictionary file with magic values

Related Skills

Technique Skills

Skill	Use Case
fuzz-harness-writing	Structure-aware fuzzing with `arbitrary` crate
address-sanitizer	Understanding ASan output and configuration
coverage-analysis	Measuring and improving fuzzing effectiveness
fuzzing-corpus	Building and managing seed corpora

Skill	Use Case
fuzzing-dictionaries	Creating dictionaries for format-aware fuzzing

Related Fuzzers

Skill	When to Consider
libfuzzer	Fuzzing C/C++ code with similar workflow
aflpp	Multi-core fuzzing or non-Cargo Rust projects
libafl	Advanced fuzzing research or custom fuzzer development

Resources

Rust Fuzz Book - cargo-fuzz Official documentation for cargo-fuzz covering installation, usage, and advanced features.

arbitrary crate documentation Guide to structure-aware fuzzing with automatic derivation for Rust types.

cargo-fuzz GitHub Repository Source code, issue tracker, and examples for cargo-fuzz.

Revision #6
Created 2026-02-18 08:40:10 UTC by John
Updated 2026-07-26 20:01:25 UTC by John