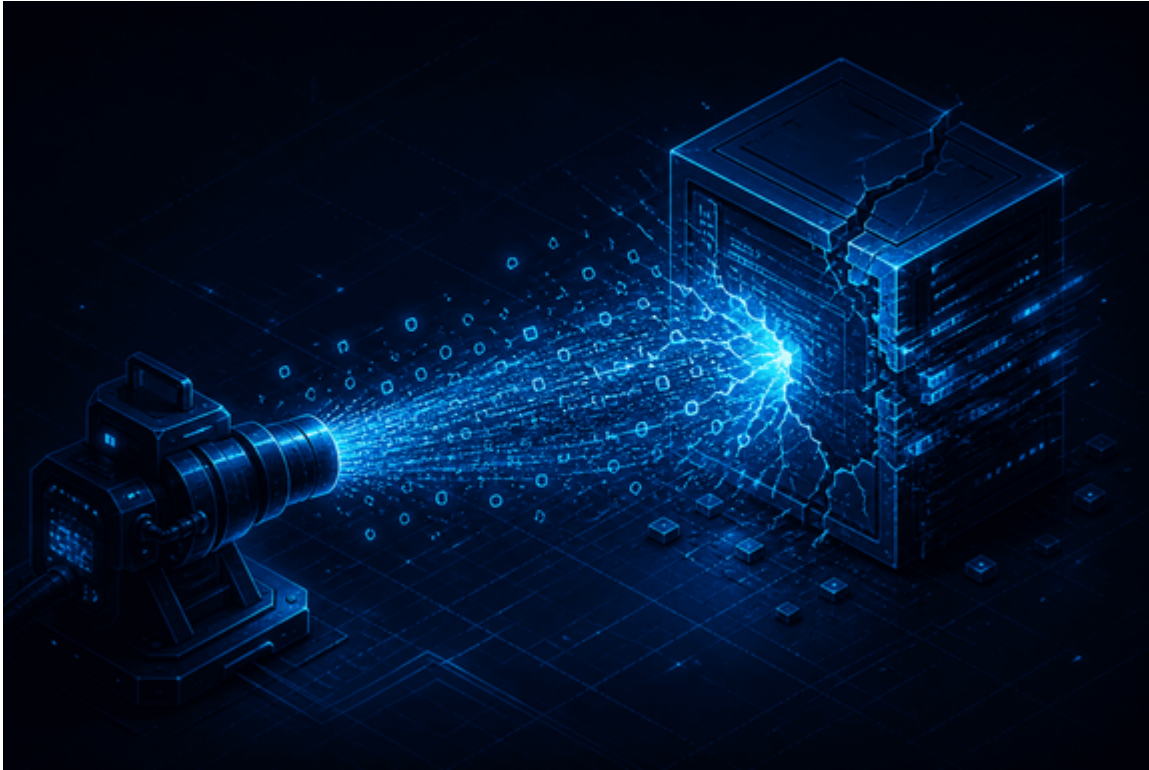


Fuzzing and Misuse Cases: Testing Like Someone Who Wants It to Break

Wed, Aug 19, 6:00 PM EDT · <https://scottslab.io/posts/fuzz-testing-and-misuse-case-testing>



TL;DR

Fuzzing throws malformed input at software until something breaks. The version that works is coverage-guided: the fuzzer instruments the binary, keeps any input that reaches new code, and evolves toward the dark corners. Run it without a sanitiser and you'll miss most memory bugs, because a corruption that doesn't crash looks like a pass. Google's OSS-Fuzz has found over 13,000 vulnerabilities and 50,000 bugs this way. Misuse case testing is the human half: it finds the logic flaws no random string will ever hit.

Most testing checks that software does the right thing with the right input. These two check what it does with the *wrong* input, on purpose. That's where the security bugs actually live.

Dumb fuzzing, and why it stopped being enough

The naive version feeds a program a flood of junk and watches for a crash. Inputs come from a list of nasty strings, a script, purely random bytes (**generation fuzzing**), or by taking real valid inputs and mangling them (**mutation fuzzing**, which usually goes deeper because it starts from something the parser already accepts).

The problem is that random bytes die at the front door. If your program parses JSON, a random byte string fails validation in the first millisecond and never reaches the interesting code. You can run that for a week and test nothing but your input validator.

Coverage-guided fuzzing, which is the actual technique

The advance that made fuzzing serious is **coverage guidance**. The fuzzer instruments the binary so it can see which code paths an input reached. If a mutated input reaches a branch nothing has reached before, it's kept and becomes a parent for the next generation. The fuzzer is now hill-climbing toward code coverage instead of guessing.

This is why it finds things humans don't. It doesn't know your format. It *discovers* it, by keeping whatever gets deeper. Given a JPEG parser and one valid JPEG, a coverage-guided fuzzer will often rediscover chunks of the file format on its own, because inputs that keep the magic bytes intact reach further and survive.

The engines worth knowing are **AFL++**, **libFuzzer** and **honggfuzz**. All three are what Google's **OSS-Fuzz** runs against open source, where the results speak plainly: **over 13,000 vulnerabilities and 50,000 bugs across around a thousand projects**.

Two practical points decide whether your run works. First, the **seed corpus**: start it with real, valid, diverse inputs, because the fuzzer evolves from what you give it and a good corpus can be the difference between hours and weeks. Second, **structure-aware fuzzing** for formats with checksums or required framing. Otherwise every mutation fails an integrity check before it reaches your logic.

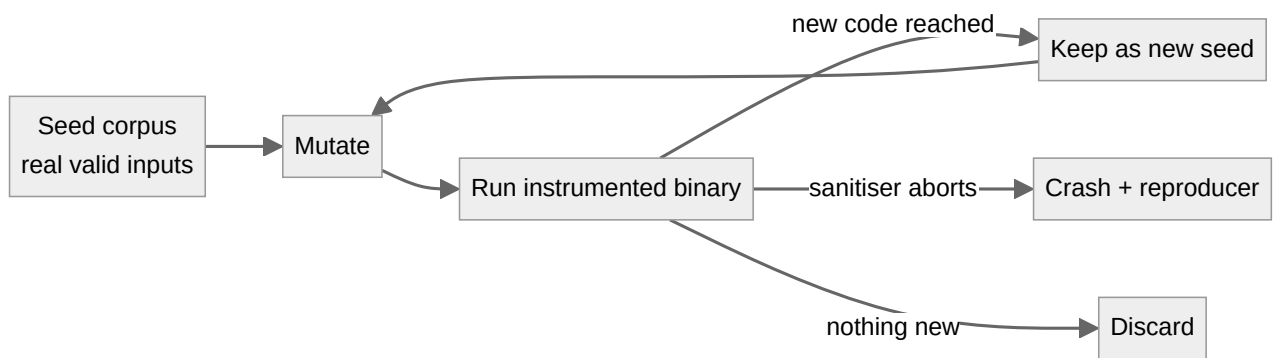
Run a sanitiser, or you're mostly wasting the electricity

This is the part that gets skipped, and it matters more than the choice of engine.

A memory bug does not reliably crash. A read a few bytes past a buffer usually returns adjacent heap memory and carries on perfectly happily. Your fuzzer sees no crash, records a pass, and moves on. The bug that leaks memory to an attacker sits right there.

Sanitisers fix that by making the failure loud. **AddressSanitizer** poisons the memory around every allocation so an out-of-bounds read aborts immediately. **UndefinedBehaviorSanitizer** catches integer overflow and the rest of the undefined-behaviour family. **MemorySanitizer** catches reads of uninitialised memory.

Fuzzing without a sanitiser finds the bugs that crash. Fuzzing with one finds the bugs that matter. Turn them on.



Run a sanitiser, or you're mostly wasting the electricity

One serious caveat that hasn't changed: fuzzing looks like an attack because it *is* one. Get written permission from the application owner before you point it at anything you don't own. Fuzzing someone else's service is how a test becomes an incident report with your name on it.

Misuse cases: the half a fuzzer can't do

Misuse case testing comes at the same goal from the design side. Instead of throwing inputs at a running system, you deliberately think like an attacker and write test cases around it. The guiding question is literally "how would someone break this?"

The difference from a penetration test is **timing**. Misuse case testing happens during development, before it ships. A pentest happens after deployment, when fixing an architectural mistake costs twenty times more. It works best with a mix of people: the developers who built it, because they know where the bodies are buried, and outside reviewers, who aren't blinded by knowing how it's *supposed* to be used.

The recurring cases are worth keeping as a checklist, because they show up everywhere:

- **Boundary violations:** the classic being whether you can debit an account below zero. Then: can you order a *negative* quantity and get refunded? Does price times quantity overflow?
- **Missing or unexpected input:** what happens when a required field is absent rather than empty?
- **Injection:** the field that ends up in a query, a shell, a template, or a log line.
- **State-order abuse:** can you skip step two and go straight to step three? Can you replay the confirmation?

That last one is the reason humans are still required. A fuzzer will never think to complete a checkout twice, because there is no malformed *input* involved. The requests are all perfectly valid. It's the *sequence* that's wrong, and pattern-matching doesn't see sequences.

The mindset, which is the actual point

Stop testing whether it works and start testing whether it can be made to fail.

Fuzzing does that mechanically, at a scale and stupidity no human can match. It will try the four-gigabyte filename you'd never bother with. Misuse cases do it deliberately, at design time, against the logic. You want both, because the machine finds the inputs you'd never think to try, and the humans find the flaws where every input was valid and the *order* was the attack.

Written by Scott S. — Contact: scott@scottschlangen.com — scottslab.io — <https://scottslab.io/posts/fuzz-testing-and-misuse-case-testing>