Fuzz testing in WebAssembly VMs

Patrick Ventuzelo
Feb 18 · 4 min read

> ***Fuzzing*** *or* ***fuzz testing*** *is an automated testing technique that involves providing invalid, unexpected, or random data as inputs to a program to find bugs that would be otherwise hard to find with manual generated input. — Wikipedia*

In the last months I've been working developing fuzzing targets to find bugs and create patches for the Wasmer WebAssembly runtime.

In this post we will learn *what is fuzzing*, *why it is important* for WebAssembly runtimes and what kind of *bugs* fuzzing helped to detect.

Quick summary

After studying Wasmer codebase using static code analysis and code review I understood the global architecture and which parts of the code should be executed

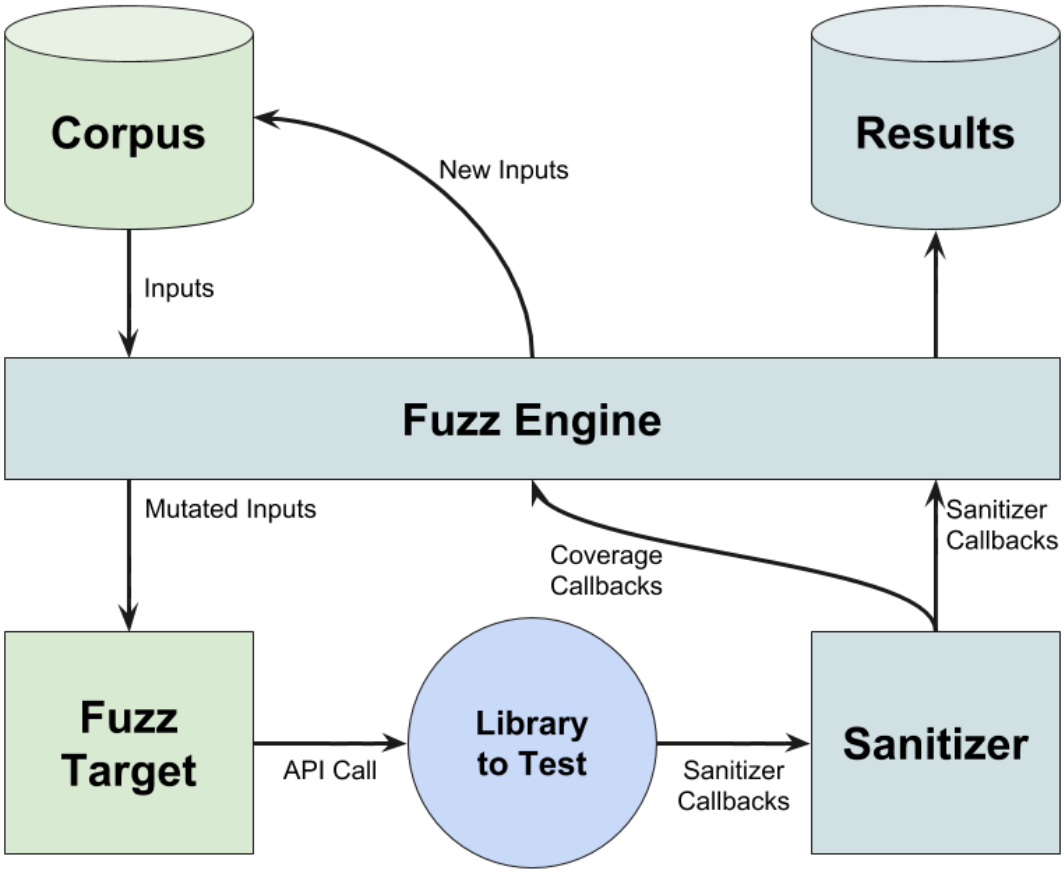
more often. Then, I developed multiple fuzz targets to test and improve the *resilience* of the Wasmer runtime.

As a result of my journey with Wasmer, around **20 issues** has been reported and fixed in the codebase including 5 bugs in external libraries that the Wasmer runtime depends on.

What is fuzzing ?

Here is a global overview of the fuzzing process:

1. You create a set of input test cases (i.e *the corpus*)
2. Fuzz engine will **pick and mutate** (*randomly*) one input sample.
3. Input is executed by Wasmer and *coverage paths* are recorded.
4. When **new paths** are reached, current input sample is **stored** in *the corpus* to be re-used later.
5. If fuzz target **crash**, crashing sample is stored on the side for later *analysis* by the user.



Fuzz testing overview with LibFuzzer. Source

During this journey I have used multiple fuzzing engines (cargofuzz, afl-rs, honggfuzz) to optimize our chance to trigger bugs since fuzzing/mutation algorithms are different for each engine.

Fuzzing is essential to find bugs, but it has also key advantages for developers.
As you have seen, during the fuzzing process, samples reaching new coverage paths are stored, meaning that your fuzzing engine will generate reusable testing samples on its own.

Why is Fuzzing important for WebAssembly runtimes?

> *WebAssembly describes a memory-safe, sandboxed execution environment that may even be implemented inside existing JavaScript virtual machines.*

As defined in official WebAssembly website, WebAssembly virtual machines (VMs) should be **Safe** and allow **Execution of untrusted code** through a *sandboxed* environment.

Fuzzing WebAssembly runtimes will allow you to detect code paths that caused:

- Runtime engines instability/crash.
- Unwanted interaction with the host system.
- Unsupported WebAssembly opcodes/features.
- Regression bugs.

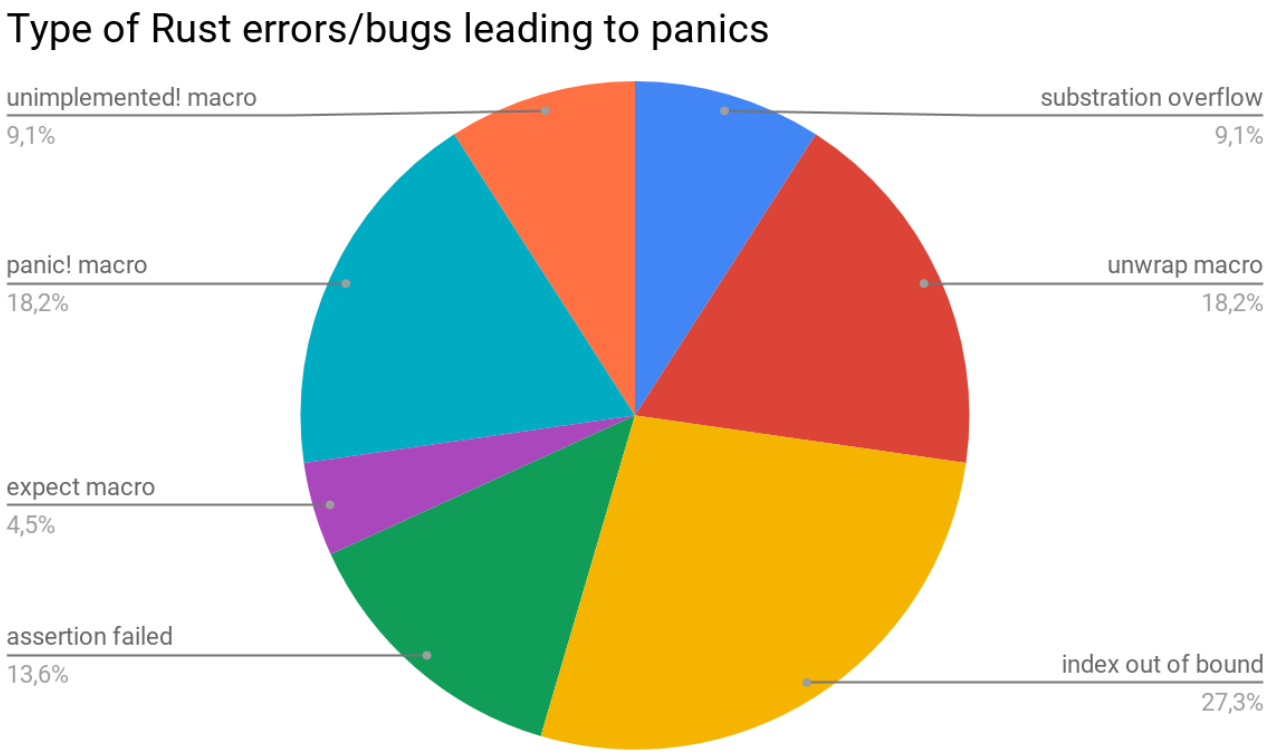
In regards to Wasmer, multiple fuzz targets has been developed to executed all available backends (LLVM, Singlepass, Cranelift) and ABIs (Emscripten, WASI) currently supported.

Which kind of bugs did we prevent?

Wasmer runtime is fully written in Rust. Even if it's a memory safe language, it doesn't mean Rust code will be exempt from bugs/vulnerabilities.

During this fuzzing journey, multiple bugs that caused Wasmer to panic have been found. The impact those bugs really depended on how you were using Wasmer instance. In the most critical case, those bugs are leading to a Denial Of Service (DoS) of the Wasmer runtime engines, which now have been fully resolved.

Here is a global overview of the type of bugs patched:



Wasmer reported bugs after fuzzing

As you can see in the following pie chart, bugs leading to panics are mainly due to *unchecked values*, or verified and caught at runtime by Rust (like index out of bounds, unwrap, substration overflow)

The other bugs leading to panics are macros and functions often used by developers during under development code like `panic!()`, `assert()`, `unimplemented!()` and `expect()`.

Community contribution

This fuzzing adventure at Wasmer also helped to find and resolve bugs in 3rd party libraries used by Wasmer like:

wasmparser: This Rust library is used for parsing WebAssembly binary files and we reported two assertion failures (#122 & #139) and one index out of bound panic (#126).

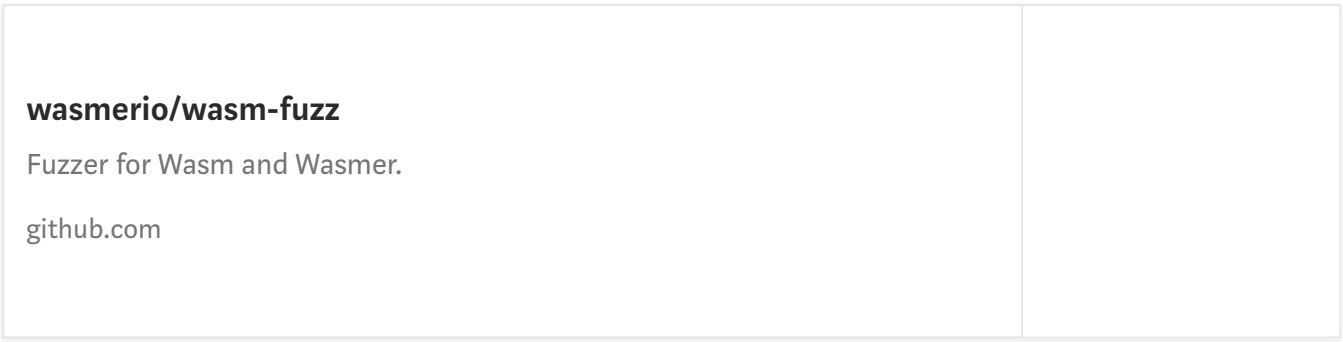
cranelift: Code generator library used as one possible backend by Wasmer to generate executable machine code (assembly) from WebAssembly bytecode. In this library, we reached a call to `panic!` macro (#1257) and triggered one assertion failure (#1306).

Conclusion

Whatever the complexity of your project code base, it's important to setup fuzzing targets during development and ideally integrate continuous fuzzing process as soon as possible.

If you are interested in learning more about this topic, I would like to invite you to check WebAssembly Security.

Here's our Wasm Fuzz repo, check it out!



Let the fuzzing journey begin! 🎉

Thanks to Mark McCaskey (declined).

Fuzzing Security Webassembly Development Testing