Smart Contract Fuzzing

How to find edge cases with echidna

Alberto Cuesta Cañada

Follow

Coinmonks

--

2

Listen

Share

Coding Smart Contracts is not for the faint of heart.

Very recently, I participated in making the Yield Protocol go live. For those that haven't heard of it, it is a fixed-rate borrowing and lending platform in Ethereum. It also implements mathematical formulas that are at the edge of what it is possible for a smart contract.

The Yield Protocol is also a deploy-and-forget platform. Once we went public we had no way of stopping users interacting with our software, and no way to fix any bugs in the core contracts.

We could go up like a rocket, and then explode.

However, we didn't. We have had a surprisingly small number of bugs, none of them in the core smart contracts.

A reason for that has been that we used a fuzzer to find edge cases that couldn't be found using unit or integration testing. Using a fuzzer is not yet a common thing to do, and many have asked me to write this article and help them fuzz their contracts as well. So here we are.

What is fuzzing?

By fuzzing, I understand testing a massive number or random scenarios, to find out values that produce unexpected results.

For example, at Yield we calculate fractional exponentials, which are calculated iteratively and can be very expensive to run on Ethereum, where every operation costs money.

To save money for our users, at Yield we hacked the exponential calculation algorithm to stop calculating decimals after a while. The downside is that our smart contracts would now transact at prices that deviated a random amount from the mathematical ideal.

But how far did the deviate? Could we set it up so that the prices remained within a dollar cent from the ideal?

The only way of knowing was to test the smart contracts with a massive number of random scenarios. Trail of Bits told us to use a fuzzer during the code audit, and that that's how we got here.

How to fuzz

First you will have to install echidna. Of the options in the website I managed to get it going downloading the precompiled binary for Ubuntu. To do that you will need to install cryptic-compile and slither as well. I remember it took me a few attempts to get it right, a rare event these days that everything comes in npm packages.

Fuzzing is not easy, the tools are rough, and the math is hard, but it is worth it.

If you need help, you should join the Empire Hacking Slack channel.

The echidna github has a lot of documentation and several tutorials that I was completely unable to absorb. What saved me was one running example from Gustavo Grieco, which I have tweaked and reused since. I will probably learn one day the finer details.

In short, you need one config.yaml

file, this is mine:

Your guesses as to what these things do are as good as mine. RTFM. I didn't.

And then you code a solidity contract with the invariants that you want to test, this one tests the muld

function from DecimalMath.sol.

We will use echidna to call theDecimalMathInvariant.muld_ function in this contract 20,000 times. Echidna knows that it has to run this function because it is public

. The function has two parameters (x and y), so echidna will use random values on those for each run.

When echidna runs, it considers reverting due to a require

to be a success, but reverting due to an assert

a failure.

In the contract above I'm checking four properties of fixed point multiplication:

1. That x * y == z, displacing the comma.
2. That DecimalMath.muld rounds down (by asserting that z / y <= x, displacing the comma).
3. That if y is greater than 1, z is greater than x (this wouldn't be true if there is an overflow).
4. That if y is smaller than 1, z is also smaller than x (this wouldn't be true if there is an underflow).

Coding invariants forces you to think how to test something by finding properties that are always held true, instead of repeating the code in the original smart contract. Finding invariants is often difficult and you should get help from the business experts when doing so.

Check the full contract to see all the invariants I'm testing. Echidna will run all the public functions in the contract.

Now you are ready to test:

Those two parameters are the name of the contract where we have coded the invariants (not the target contract that we are testing, and it is not a path either). The second argument is a path to the .yaml file from before.

Echidna will spend some time analysing the contract and getting set up, and eventually you'll see this:

You might notice something strange in there. What is this UNIT

that we are fuzzing?

Turning invariants on and off

Now, there surely is a way of doing this in theyaml file, but I use the visibility of the functions to turn invariants on and off.

Echidna will run fuzzing on every public

function in your contract. That's probably not what you want. Sometimes you might want to test just a function that is causing trouble, instead of the 10x in the contract, of which 9 pass. Or sometimes you might have some helper functions that you don't want to fuzz.

In those cases you can just declare the function internal

and echidna will ignore it. Declare all your state variables internal

as well, or they will get uselessly fuzzed.

In this file I have a helper function to help me calculate the invariant in the Yield whitepaper. I will use that function everywhere else, and I don't want to test it on its own, because I have no way to confirm it works out of context.

In the same file I even have an invariant testing function that is commented out by making it internal

. That's left from development and should have been cleaned up, how embarrasing.

At the top you will see that there are a bunch of constants, all of them internal. If they would be public echidna would consider them functions and fuzz them, same as happened with UNIT

, which was inherited from a parent contract and happened to be public

, so it got fuzzed.

These were the first fuzzing contracts I wrote, which is fortunate because being libraries or stateless contracts they were

easy to fuzz. Shortly after I started working on fuzzing stateful contracts, and that taught me I was doing everything quite wrong.

Fuzzing stateful contracts

I soon needed to fuzz contracts that keep a state, instead of just mathematical libraries. Thinking about it and poking around I realized that the tests that echidna runs are not run in isolation.

Echidna runs the functions in the contract randomly, until something fails. I think that the seqLen parameter in the .yaml file must define how many function calls are in a sequence.

Within a sequence, state is maintained. Instead of having the invariant contract inherit from the contract to test, it works better to link them together like in this example for WETH10.

Before I was testing each function in isolation, but now I keep the state instead. Echidna will execute functions in WETH10, looking for failed assertions. My invariant-testing functions now look like this:

With this, I'm testing that whatever the state WETH10 was in before, supply and balances change as they should with deposit

and withdraw

. You can use this to build more complex multi-contract setups.

Debugging the fuzz

It's great when all the tests pass. Unless that happens the first time, because then you should be suspicious and assume you coded the tests wrong. So first you need to get your tests to fail.

When tests fail, Echidna will find a combination of function calls that makes an assertion fail. Then it will work out the shortest sequence that causes that failure, often bringing it down to one or two calls. Echidna will tell you the function calls it executed and that's it.

You won't get logs, information on which specific assert failed, or content of state variables. It's not going to be easy to know why something failed.

You will need to run the sequence in the truffle console, or in a regular test file which sometimes is more comfortable, so that you can poke the smart contract around with that specific case, and start investigating why the assert failed.

If echidna tells me that my "buy Dai and reverse the trade" invariant fails with certain parameters, I paste them into the test file that I use as a companion to the fuzzing contract. Then I can get a dump of all the relevant variables and start debugging in a traditional setting.

Conclusion

Fuzzing is not easy, the tools are rough, and the math is hard, but it is worth it.

Fuzzing gives me a level of confidence in my smart contracts that I didn't have before. Relying just on unit testing anymore and poking around in a testnet seems reckless now.

That's about it. That much I know. It's not a lot, but for me it has worked out well so far. Thanks a lot for getting this far, and good luck!

Also, Read

- The Best Crypto Trading Bot
- Crypto Copy Trading Platforms
- The Best Crypto Tax Software
- Best Crypto Trading Platforms
- Best Crypto Lending Platforms
- Best Blockchain Analysis Tools
- Crypto arbitrage guide: How to make money as a beginner
- Best Crypto Charting Tool
- Ledger vs Trezor

- What are the best books to learn about Bitcoin?
- 3Commas Review
- AAX Exchange Review | Referral Code, Trading Fee, Pros and Cons
- Deribit Review | Options, Fees, APIs and Testnet
- FTX Crypto Exchange Review
- NGRAVE ZERO review
- Bybit Exchange Review
- 3Commas vs Cryptohopper
- The Best Bitcoin Hardware wallet
- Best monero wallet
- ledger nano s vs x
- Bitsgap vs 3Commas vs Quadency
- Ledger Nano S vs Trezor one vs Trezor T vs Ledger Nano X
- BlockFi vs Celsius vs Hodlnaut
- Bitsgap review — A Crypto Trading Bot That Makes Easy Money
- Quadency Review- A Crypto Trading Bot Made For Professionals
- PrimeXBT Review | Leverage Trading, Fee and Covesting
- Ellipal Titan Review
- SecuX Stone Review
- BlockFi Review | Earn up to 8.6% interests on your Crypto