This article teaches how to write Solidity smart contrac fuzz tests (fuzzing) to help your writing more secure protocols and uncover issues in your codeo.

Smart contracts fuzzing is the new floor for smart contract security and a must for any developer before deploying code to a blockchain. Plenty of tools are available on the market to perform fuzzing, and in today's blog, we will delve into Fuzzing with Foundry.

For those who don't know what Solidity smart contract Fuzzing Invariant testing is, don't forget to check out our specialized article explaining this technique's nuances, with simple examples and analogies.

If you haven't ever coded a single line of code, check out our UltimateBlockchain Developer course from zero to expert on Updraft.

Before getting started with smart contract fuzzing, let's quickly understand what an invariant is, as it's a key component to keep in mind when setting up our fuzz tests.

Defining invariant in your Solidity smart contract

An invariant

is a condition that the system must always hold, regardless of the contract's state or input.

In DeFi, a good invariant might be:

- The number of votes cast must not exceed the number of registered voters.
- A user should never be able to withdraw more money than they deposited.
- There can only be one winner of the fair lottery.

Foundry defines an invariant test as a stateful fuzz test.

Still, this definition is not entirely accurate as we can perform any test to an invariant

, as seen in the following sections.

Stateless Solidity smart contract fuzz testing

To perform a stateless fuzz test on your Solidity smart contract using Foundry, on which the state of the variables will be forgotten on each run, let's consider a simple example - If you want to follow along with the code, you can start a new Foundry project.

Now, let's create a smart contract called SimpleDapp that will allow users to deposit and withdraw funds from the protocol and the contract looks as follows:

The invariant

, or the property that the system must always hold for this contract, is: a user should never be able to withdraw more money than they deposited.

Setting up a Stateless fuzz test for this Solidity contract is easy in Foundry, and we need to create a test as we always do on this framework.

First, we need to use the basic imports and define a setup function

Then, we can easily set up a test for this contract. The key to making them a stateless fuzz test

is that instead of using the test parameters burned on the code, we set them up as input parameters; this way, Foundry will automatically start throwing random input data at them.

For this test, the fuzzer will try random values for both variables depositAmount and withdrawAmount. If the withdrawal amount exceeds the deposited amount, the test will fail; let's try it using the command:

As expected, this will throw us an error stating that the invariant condition was violated as all the deposits and withdrawals are random; there will be a scenario in which the withdrawal value will be greater than the deposited one.

As you can see, together with the number used to break our function, there's another parameter: "runs" - this represents the number of randomly generated inputs the fuzzer went through before it found the CounterExample

. If the fuzzer tests thousands of potential counterexamples

and none of them work because there isn't a bug, then you might end up waiting eternally.

To solve this, we can set up the maximum amount of runs the fuzzer will try before stopping; in Foundry, we need to access the configuration file foundry.toml

.

Then, we can set up a new parameter called [fuzz]

and manually state the maximum amount of runs. The end result will look something like this.

Stateful Solidity smart contract fuzz testing

For this type of test, the state of the variables is remembered across multiple runs, and we need to go through some unique configurations on Foundry to make this work.

Let's explore a different example for a new contract called AlwaysEven.sol

; this time, we have set up an invariant for a variable called alwaysEvenNumber

, and the condition that it must always hold is that the variable must always be even, never odd.

So, the contract is as follows.

We also include another variable called hiddenValue

, which can change the value of alwaysEvenNumber

to the odd value of three. As the state of the variables will be remembered, this will most likely break the invariant condition.

Setting up the Stateful Test

First, we need an extra import from the standard forge-std

library as follows:

We need to add this as part of the inheritance of our test contract like this:

From the StdInvariant.sol

we get a new function called targetContract

. This will allow us to define the contract we will put to the test.

The exciting part of this is that by defining a target contract, Foundry will automatically start executing all the contract functions randomly and setting random input parameters as well. To define the target contract, we need to set it up on the setup

function.

Finally, we can set up the test. This time, we don't need to include an input parameter for the test, and as the function will be executed automatically, we need the assertion statement. The final result would look like this:

When the fuzzer starts throwing random data to the functions, it will eventually set an input parameter of 8, ultimately making the invariant condition break and resulting in an error. Let's run the test using the command:

We will have a result like this one.

A note on fuzz testing terminology for Solidity smart contracts

When it comes to different types of tests, terminology can be confusing. Foundry often categorizes an invariant test as a stateful fuzz test

, even though we can perform any test using an invariant

, from unit tests to any fuzz test.

To clarify these distinctions, here's a detailed graph by Nisedo outlining the various test types.

So, remember that you can define an invariant

- or property that the system must always hold and perform any test you want to it. Foundry requires you to use the keyword invariant

for stateful invariant fuzz testing, but that does not mean that it is the only type of invariant testing.

Conclusion

Adopting fuzz and invariant testing transcends standard practice in smart contract development—it's a vital necessity.

We hope you enjoyed this guide to performing Solidity smart contract invariant fuzz testing using the Foundry framework. Plenty of tools exist, but Foundry stands on the top as it allows quick development of smart contracts.

If you want to tinker with this code, don’t forget to check out the contract source code on GitHub.