You can watch the video for this blog here.

Smart contracts have been tested and audited, but are they truly bulletproof? Just when you think your code is secure, an attacker might exploit a vulnerability that you never even thought about. To ensure maximum security, contracts need to withstand not just individual attack scenarios but every possible scenario. That's where fuzz testing comes in.

Fuzz testing, or fuzzing, involves providing random data to a system with the intent to locate vulnerabilities or errors. In this blog post, we'll break down the basics of fuzz testing for smart contracts, and then we'll build on those foundations to explore some more advanced concepts.

The Power of Fuzz Testing

Let's say we have a smart contract function called doStuff

that takes an integer as an input parameter. The function guarantees that a certain variable, shouldBeZero

, is always set to zero. This characteristic, known as an invariant or property, should always hold true regardless of the input provided to the function.

For such cases, we can write a fuzz test that generates random input data and checks whether the invariant holds true. This is where fuzz testing shines: it stresses the system with various inputs and combinations to uncover scenarios that might breach the invariant, thereby exposing vulnerabilities in the code. To understand fuzz testing better, let's dive into the types of fuzz testing and how to write them.

Stateless Fuzz Testing

Stateless fuzz testing sends random data to an input with each test, discarding the results of the previous test. Here's a simple example of a stateless fuzz test written in Foundry:

data = ... # Randomly generate data contract.doStuff(data) assert contract.shouldBeZero() == 0

This test will call the doStuff

function with random data and verify that shouldBeZero

is, indeed, zero after the call. With tools likeTrail of Bits Echidna and Optic, it's possible to automate the process of generating random data and running multiple tests.

However, stateless fuzz tests may miss vulnerabilities that only become apparent when a specific sequence of function calls with specific inputs occur. This leads us to another type of fuzz testing: stateful fuzz testing.

Stateful Fuzz Testing

Stateful fuzz testing addresses scenarios where the state of the system is affected by a series of function calls. Unlike stateless fuzz testing that discards the previous test's state, stateful fuzz tests keep track of changes made by previous calls. In Foundry, we can use the invariant

keyword to write stateful fuzz tests. Here's how we set up a stateful fuzz test in Foundry:

1. Import the StdInvariant

contract and inherit it in our test contract.

1. Specify which contract and functions the fuzzer should call randomly.

target_contract(contract_address) # Address of the example contract

1. Write a new function with the invariant

keyword to check whether the invariant holds true after a series of random function calls.

function invariant_test_always_is_zero() public { assert exampleContract.shouldBeZero() == 0; }

Using stateful fuzzing, we can uncover vulnerabilities that may not have been detected with stateless fuzz testing alone.

Making Fuzz Testing the Norm

Implementing fuzz testing is a vital step to ensure the security and robustness of smart contracts. By understanding the invariants of a system and using fuzz testing to validate them, developers can prevent potential attacks and create a safer Web 3 ecosystem.

If you're working with a protocol that isn't utilizing fuzz testing, don't hesitate to raise a red flag and initiate change. Be proactive and ensure your smart contracts are well-protected from vulnerabilities.

Fuzz testing is not a silver bullet, and it cannot replace the need for expert manual review. But by incorporating fuzz testing alongside other security measures, developers can significantly enhance the safety and reliability of their smart contracts.

Going Beyond the Basics

This blog post only scratches the surface of fuzz testing's potential. From advanced fuzz testing strategies to more sophisticated invariant tests, there's so much more to learn. Stay tuned for our next blog post, where we'll dive deeper into fuzz testing and take your security efforts to the next level. Together, let's make Web 3 safer and more secure!