TRAIL OF BITS

Echidna Workshop: Part 1/6 - The Basics

Before starting

- **git clone https://github.com/crytic/echidna-streaming-series**
- **Open the part1/ folder**

Download Echidna: https://github.com/crytic/echidna

Upcoming workshops

Beginner

- **Part 1: The Basics (today)**
- Part 2: Fuzzing arithmetics and functions (Nov 22, 2022)

Intermediate

- Part 3: Introduction to AMM's invariants (Nov 30, 2022)
- Part 4: AMM fuzzing (Dec 6, 2022)

Advanced

- Part 5: Introduction to advanced DeFi's invariants (Dec 14, 2022)
- Part 6: Advanced DeFi invariants (Dec 21, 2022)

Who am I?

Anish Naik, Security Engineer I

(@anishrnaik)

Who You Should Follow

- **Troy Sargent (@0xalpharush)**
- **Josselin Feist (@montyly)**
- **Nat Chin (@0xicingdeath)**
- **Justin Jacob (@technovision99)**

Who are we?

Trail of Bits (@trailofbits)

- We help developers to build safer software
- R&D focused: we use the latest program analysis techniques
- Slither, Echidna, Tealer, Amarna, solc-select, ..

Today's agenda

- **How to find bugs?**
- **What is fuzzing?**
- **Interactive exercises**
- **How to define good invariants?**
- **Conclusion**

How to find bugs?

How to find bugs?

```
/// @notice Allow users to buy token. 1 ether = 10 tokens
/// @param tokens The numbers of token to buy
/// @dev Users can send more ether than token to be bought, to give gifts to the
team.
function buy(uint tokens) public payable{
    _valid_buy(tokens, msg.value);
    _mint(msg.sender, tokens);
}


/// @notice Compute the amount of token to be minted. 1 ether = 10 tokens
/// @param desired_tokens The number of tokens to buy
/// @param wei_sent The ether value to be converted into token
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```


There are 4 main testing techniques

- **Unit testing**
- **Manual analysis**
- **Fully automated analysis**
- **Semi-automated analysis**

Unit testing

- **Benefits**
  - Well understood by developers
- **Limitations**
  - Mostly cover “happy paths”
  - Might miss edge cases

An example unit test

```
function test_buy() public {
    uint256 tokens_to_receive = 1_000e18;  // setup parameter
    uint256 ether_to_send = 100e18;  // setup parameter
    uint256 pre_buy_balance = token.balanceOf(address(this));
    mock.buy.call{value: ether_to_send)(tokens_to_receive);
    assert(token.balanceOf(address(this)) == pre_buy_balance + tokens_to_receive)
}
```


Manual review

- **Benefits**
  - Can detect any bug
- **Limitations**
  - Time consuming
  - Require specific skills
  - Does not track code changes

Example: Security audit

Fully automated analysis

- **Benefits**
  - Quick & easy to use
- **Limitations**
  - Cover only some classes of bugs

Example: Slither

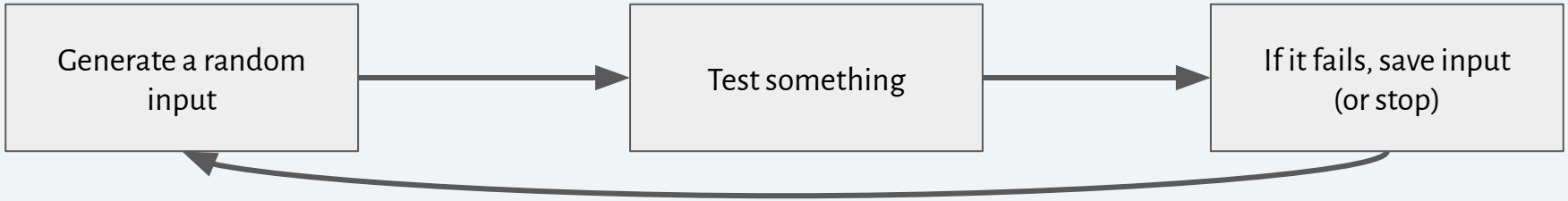
Semi-automated analysis

- **Benefits**
  - Great for logic-related bugs
- **Limitations**
  - Require human in the loop

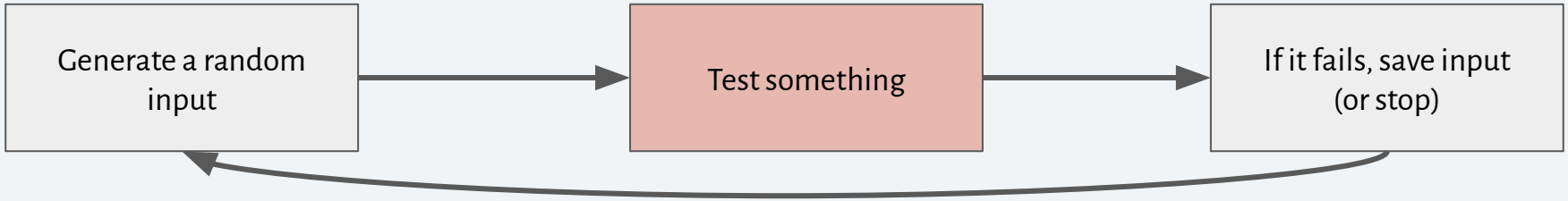
Example: Fuzzing with Echidna!

What is fuzzing?

What is fuzzing?



What is fuzzing?



Traditional fuzzing

- **Fuzzing is well established in traditional software security**
  - AFL, Libfuzzer, go-fuzz, ..
- **Stress the program with random inputs**

Goal: Crash the program!

Property based testing

- **Smart contracts don't (really) have crashes**
- **User defines some properties (a.k.a invariants)**
- **Fuzzer generates random inputs**
- **Check whether specified "incorrect" state can be reached**

Goal: Validate Properties!

What is an invariant / property?

- “Invariant Testing” = “Property Testing”
- System properties should always be true *given some pre-conditions*

System properties highlight the expected behavior of the system!

invariant adjective

Save Word

in·vari·ant | \ (ˌ)in-ˈver-ē-ənt \

Definition of *invariant*

: CONSTANT, UNCHANGING

specifically : unchanged by specified mathematical or physical operations or transformations

// *invariant* factor

Example invariant

User balance never exceeds total supply

Echidna is a smart contract fuzzer

- **Tests system properties**
- **Open source: github.com/crytic/echidna**
- **Heavily used in audits & mature codebases**

Public use of Echidna

Property testing suites

This is a partial list of smart contracts projects that use Echidna for testing:

- Uniswap-v3
- Balancer
- MakerDAO vest
- Optimism DAI Bridge
- WETH10
- Yield
- Convexity Protocol
- Aragon Staking
- Centre Token
- Tokencard
- Minimalist USD Stablecoin

Echidna's inputs: target contract + properties to test

Smart Contract Code

```
contract Token {
        uint256 totalSupply;
        mapping (address => uint256) balances;
        function transfer(address to, uint256
amount) {
        }
}
```

Property Invariant

```
function echidna_total_supply() public returns(bool)
{
    return balances[msg.sender] <= totalSupply()
}
```

input

Echidna Tests

Can Echidna break the invariant?

Exercises

Key considerations

1. **Think of Echidna as an externally-owned account (EOA)**
2. **Echidna will do the following:**
    a. Call a **sequence** of functions with random inputs in the target *and* inherited contracts
    b. Check whether the property holds
3. **Properties just resolve to a “truthy” value**

Exercise 1

- exercise1/README.md
- **Goal: check for correct arithmetic**
- **Note: use Solidity 0.7 (see solc-select if needed)**

Exercise 1 - Template

```
contract TestToken is Token {

    address echidna_caller = msg.sender;



    constructor() public {
        balances[echidna_caller] = 10000;
    }


    // add the property

}
```

Exercise 1 - Solution

```
contract TestToken is Token {

    address echidna_caller = msg.sender;


    constructor() public {
        balances[echidna_caller] = 10000;
    }


    function echidna_test_balance() view public returns(bool) {
        return balances[echidna_caller] <= 10000;
    }
```

Exercise 1 - Solution

```
$ echidna-test solution.sol
```

```
echidna_test_balance: FAILED! with ReturnFalse

Call sequence:
1.transfer(0x0,10093)
```

Exercise 2

- exercise2/README.md
- **Goal**: check for correct access control of the token

Exercise 2 - Template

```
contract TestToken is Token {

    constructor() {
        paused();
        owner = 0x0; // lose ownership
    }

    // add the property
}
```


Exercise 2 - Solution

```
contract TestToken is Token {

    constructor() {
        paused();
        owner = 0x0; // lose ownership
    }

    function echidna_no_transfer() view returns(bool) {
        return is_paused == true;
    }
}
```


Exercise 2 - Solution

```
$ echidna-test solution.sol
```

```
echidna_no_transfer: FAILED! with ReturnFalse

Call sequence:
1.Owner()
2.resume()
```

How to define good invariants

Defining good invariants

- **Start small, and iterate**
- **Steps**
  1. Define invariants in English
  2. Write the invariants in Solidity
  3. Run Echidna
     - If invariants broken: investigate
     - Once all the invariants pass, go back to (1)

There are two types of invariants

- **Function-level invariant**
  - Doesn't **rely on much of the system** OR could be **stateless**
  - Can be tested in an **isolated** fashion
  - Examples: Associative property of addition OR depositing tokens in a contract
- **System-level invariant**
  - Relies on the **deployment** of a **large part or entire system**
  - Invariants are usually **stateful**
  - Examples: user's balance < total supply OR yield is monotonically increasing

Example of a function-level invariant

- **Inherit the targets**
- **Create function and call the targeted function**
- **Use `assert` to check the property**

```
contract TestMath is Math{
    function test_commutative(uint a, uint b) public {
        assert(add(a, b) == add(b, a));
    }
}
```

Example of a system-level invariant

```
contract TestToken is Token {
    address echidna_caller = msg.sender;

    constructor() public{
        balances[echidna_caller] = 10000;
    }

    function test_balance() public{
        assert(balances[echidna_caller] <= 10000);
    }
}
```

Testing system level invariants require initialization

- **Simple initialization**
  - Deploy everything in the constructor
- **Complex initialization**
  - Leverage your unit tests framework with etheno

NOTE: Function-level invariants may also need *some* system initialization

Revisiting the original example

So how can Echidna find the bug?

```
/// @notice Allow users to buy token. 1 ether = 10 tokens
/// @param tokens The numbers of token to buy
/// @dev Users can send more ether than token to be bought, to give gifts to the
team.
function buy(uint tokens) public payable{
    _valid_buy(tokens, msg.value);
    _mint(msg.sender, tokens);
}


/// @notice Compute the amount of token to be minted. 1 ether = 10 tokens
/// @param desired_tokens The number of tokens to buy
/// @param wei_sent The ether value to be converted into token
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```


Start testing with _valid_buy

- **buy is stateful**
- **_valid_buy is stateless**
  - Start with it

What invariants exist for this function?

```
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```

Invariant: If wei_sent is zero, desired_tokens must be zero

```
function assert_no_free_token(uint desired_amount) public {
     require(desired_amount>0);
     _valid_buy(desired_amount, 0);
     assert(false); // this should never be reached
}
```

Echidna finds the bug!

```
Tests
assertion in assert_no_free_token(uint256): FAILED! with ErrorUnrecognizedOpc

Call sequence:
1.assert_no_free_token(1)
```

Conclusion

Key takeaways

- **Property-based testing aids in identifying logical flaws in a codebase**
- **Echidna's inputs are a target contract and one or more "properties" to test**
- **Echidna outputs whether or not any of the properties failed to be true**
- **Properties are simply boolean statements**
- **Can test properties in "property" mode or "assertion mode"**

Let’s fuzz test ABDK Math next week!

- Learn about the ABDK64x64 math library
- Write fuzz tests for the library
  - Function-level testing
  - Test mathematical properties (symmetric property, commutative property, etc.)
- Debug and optimize your fuzz tests

Your homework assignment

Your homework assignment for next week :)

- **Complete exercises 3-6 in building-secure-contracts**
  - https://github.com/crytic/building-secure-contracts/tree/master/program-analysis/echidna
- **Start reviewing ABDKMath64x64.sol**
  - https://github.com/abdk-consulting/abdk-libraries-solidity/blob/master/ABDKMath64x64.sol
- **Join Empire Hacking and ask me questions!**
  - empirehacking.slack.com
  - Channel #ethereum

Thanks for watching!