TRAIL OFBITS

Building secure contracts: How to fuzz like a pro

Who are we?

- Nat Chin (@0xicingdeath)
- Josselin Feist (@montyly)

- Trail of Bits: <u>trailofbits.com</u>
 - We help developers to build safer software
 - R&D focused: we use the latest program analysis techniques
 - Slither, Echidna, Tealer, Amarna, solc-select, ...

Agenda

- How to find bugs?
- What is property based testing?
- How to define good invariants?
- Where to focus?
- Comparison with similar tools

Goal: understand how to leverage fuzzing to write better code

https://github.com/trailofbits/publications/pull/197

```
/// @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);
}
```

• 4 main techniques

- Unit tests
- Manual analysis
- Fully automated analysis
- Semi automated analysis

Unit tests

- Benefits
 - Well understood by developers
- Limitations
 - Mostly cover "happy paths"
 - Might miss edge cases

How to find bugs?

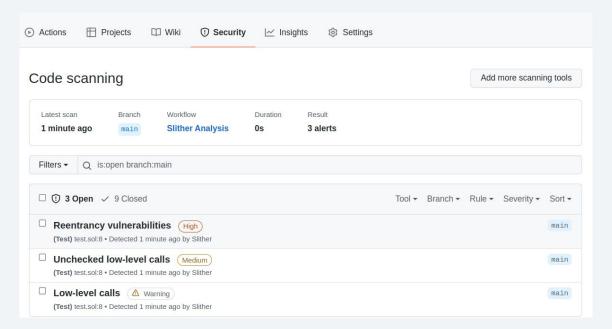
```
function test_buy(uint256 tokens_to_receive, uint256 ether_to_send) public {
    uint256 pre_buy_balance = token.balanceOf(address(this));
    mock.buy.call{value: ether_to_send)(tokens_to_receive);
    assert(token.balanceOf(address(this)) == pre_buy_balance + tokens_to_receive)
}
```

Manual review

- Benefits
 - Can detect any bug
- Limitations
 - Time consuming
 - Require specific skills
 - Does not track code changes
- Ex: Security audit

- Fully automated analysis
 - Benefits
 - Quick & easy to use
 - Limitations
 - Cover only some class of bugs
 - Ex: Slither

Slither Action

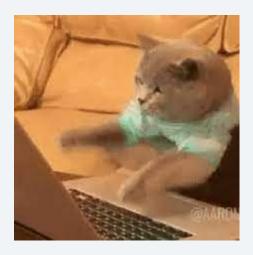


- Semi automated analysis
 - Benefits
 - Great for logic-related bugs
 - Limitations
 - Require human in the loop
 - Ex: Property based testing with <u>Echidna</u> (today's topic)

What is property based testing?

Fuzzing

- Stress the program with random inputs
 - Most basic fuzzer: randomly type on your keyboard
- Fuzzing is well established in traditional software security
 - o AFL, Libfuzzer, go-fuzz, ...



Property based testing

- Traditional fuzzer usually for crashes
 - Smart contracts don't (really) have crashes
- Property based testing
 - User defines invariants
 - Fuzzer generates random inputs to check the invariants
 - "Unit tests on steroids"

Invariant

 Something that must always be true

invariant adjective



Definition of *invariant*

: CONSTANT, UNCHANGING

specifically: unchanged by specified mathematical or physical operations or transformations

// invariant factor

Echidna

- Smart contract fuzzer
- 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

Invariant - Token's total supply

User balance never exceeds total supply

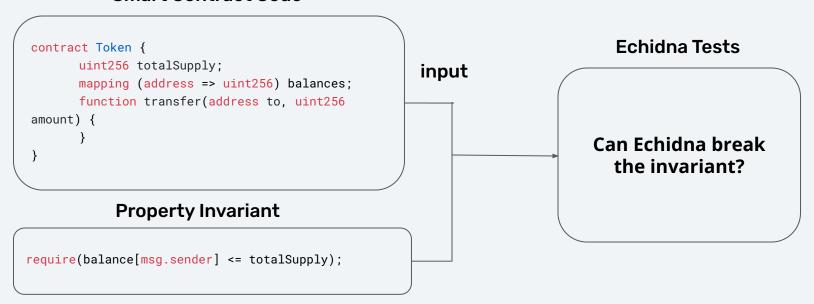
Echidna - Workflow

- Write invariant as Solidity code
- "User balance never exceeds total supply"

```
function echidna_balance_of_total_supply() public
returns(bool){
    return balanceOf(msg.sender) <= _totalSupply;
}</pre>
```

Echidna - Workflow

Smart Contract Code



Echidna - Demo

```
pragma solidity 0.7.0;

contract Token{
    mapping(address => uint) public balances;
    function transfer(address to, uint value) public{
        balances[msg.sender] -= value;
        balances[to] += value;
    }
}
```

Echidna - Demo

```
pragma solidity 0.7.0;
contract TestToken is Token {
    address echidna_caller = msg.sender;
    constructor() public {
        balances[echidna_caller] = 10000;
    // the property
    function echidna_test_balance() public view returns (bool) {
        return balances[msg.sender] <= 10000;</pre>
```

Echidna - Demo

```
Tests found: 1
Seed: -8490366082558344445
Unique instructions: 150
Unique codehashes: 1
Corpus size: 1

echidna_test_balance: FAILED! with ReturnFalse

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

Campaign complete, C-c or esc to exit
```

https://github.com/crytic/building-secure-contracts/blob/master/program-analysis/echidna/Exercise-1.md

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)

Identify invariants

- Sit down and think about what the contract is supposed to do
- Write the invariant in plain
 English



Identify invariants: Maths

Math library

- Commutative property
 - 1+2=2+1
- Identity property
 - **■** 1 * 2 = 2
- Inverse property
 - = x + (-x) = 0

Identify invariants: tokens

- ERC20.total_supply
 - No user should have a balance > total_supply
- ERC20.transfer:
 - After calling transfer
 - My balance should have decreased by the amount
 - The receiver's balance should have increased by the amount
 - If the destination is myself, my balance should be the same
 - If I don't have enough funds, the transaction should revert/return false

Write invariants in Solidity

- Identify the target of the invariant
 - Function-level invariant
 - Ex: arithmetic's associativity
 - Usually stateless invariants
 - Can craft scenario to test the invariant.
 - System-level invariant
 - Ex: user's balance < total supply
 - Usually stateful invariants
 - All functions must be considered

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));
    }
}
```

System level invariant

- Require initialization
 - Simple initialization: constructor
 - Complex initialization: leverage your unit tests framework with etheno
- Echidna will explore all the other functions

System level invariant

```
contract TestToken is Token {
    address echidna caller =
0x00a329C0648769a73afAC7F9381e08fb43DBEA70;
    constructor() public{
        balances[echidna_caller] = 10000;
    function test_balance() public{
        assert(balances[echidna_caller] <= 10000);
```

- In practice: you don't know where the bugs are
- Code coverage vs behavior coverage
 - Cover as many functions as possible or;
 - Focus on specific components?

Try different strategies

- Behavior coverage first
 - Focus on 1 or 2 components
- Code coverage first
 - Cover many functions with simple properties
- Alternate: 1 day on behavior coverage, then 1 day on code coverage,

• • •

No right or wrong approach: try and see what works for you

- Start simple, then think about composition, related behaviors, etc...
 - Can transfer and transferFrom be equivalent?
 - transfer(to, value) ?= transferFrom(msg.sender, to, value)
 - o Is transfer additive-like?
 - transfer(to, v0), transfer(to, v1) ?= transfer(to, v0 + v1)?

Where to focus?

- Start simple, then think about composition, related behaviors, etc...
 - Can transfer and transferFrom be equivalent?
 - transfer(to, value) ?= transferFrom(msg.sender, to, value)
 - o Is transfer additive-like?
 - transfer(to, v0), transfer(to, v1) ?= transfer(to, v0 + v1)?
 - Spoiler: this won't hold; why?

Where to focus?

- Building your own experience will make you more efficient over time
- Learn on how to think about invariants is a key component to write better code

```
/// @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);
}
```

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

• What invariants?

```
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);
}
```

- What invariants?
 - If wei_sent is zero, desired_tokens must be zero

```
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);
}
```

```
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
}
```

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

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

Comparison with similar tools

Other fuzzers

- Inbuilt in dapp, brownie, foundry, ...
- Might be easier for simple test, however
 - Less powerful (e.g. not stateful in foundry)
 - Require specific compilation framework

Formal methods based approach

- Manticore, KEVM, Certora, ...
- Provide proofs, however
 - More difficult to use
 - Return on investment is significantly higher with fuzzing



Echidna's advantages

- Echidna has unique additional advanced features
 - Can target high gas consumption functions
 - Differential fuzzing
 - Works with any compilation framework
 - Different APIs
 - Boolean property, assertion, dapptest/foundry mode, ...
- Free & open source

Conclusion

Conclusion

- https://github.com/crytic/echidna
- To learn more: github.com/crytic/building-secure-contracts
- Start by writing invariants in English, then write Solidity properties
 - Start simple and iterate
- Your mission
 - Try Echidna on your current project

ToB is hiring (https://jobs.lever.co/trailofbits)

Security Consultants & Apprentices