TRAIL OF BITS

Fuzzing like a security engineer

Who am I?

- **Nat Chin (@0xicingdeath)**

- **Trail of Bits: trailofbits.com**
  - 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 do we find bugs?
- What is property based testing?
- How to define good invariants?
- Comparison with similar tools

Goal: understand how to leverage fuzzing to write better code

Let’s start

```
git clone
https://github.com/crytic/building-secure-contracts.git

git checkout eth-taipei-workshop
```

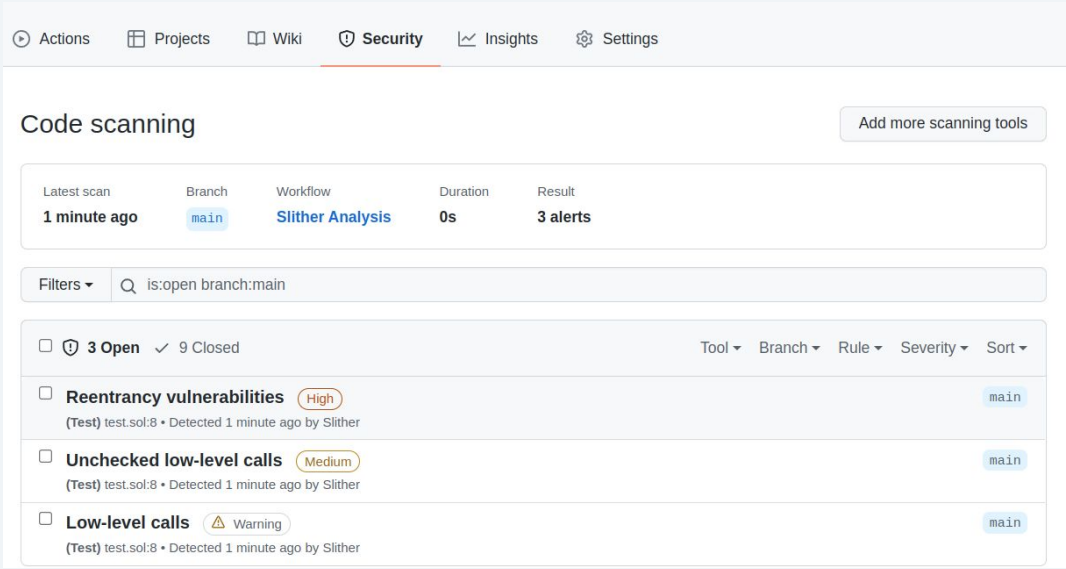
How do we find bugs?

- **Unit testing**
- **Manual Analysis**
- **Automated Analysis – fully automated or semi automated**

Fully automated

- **Results in many findings**
- **Usually requires manual triaging**

Full automated - Example



Semi-automated analysis

- **Benefits**
  - Great for logic-related bugs
- **Limitations**
  - Require human in the loop
- **Ex: Property based testing with Echidna (today's topic)**

What is property based testing?

Fuzzing

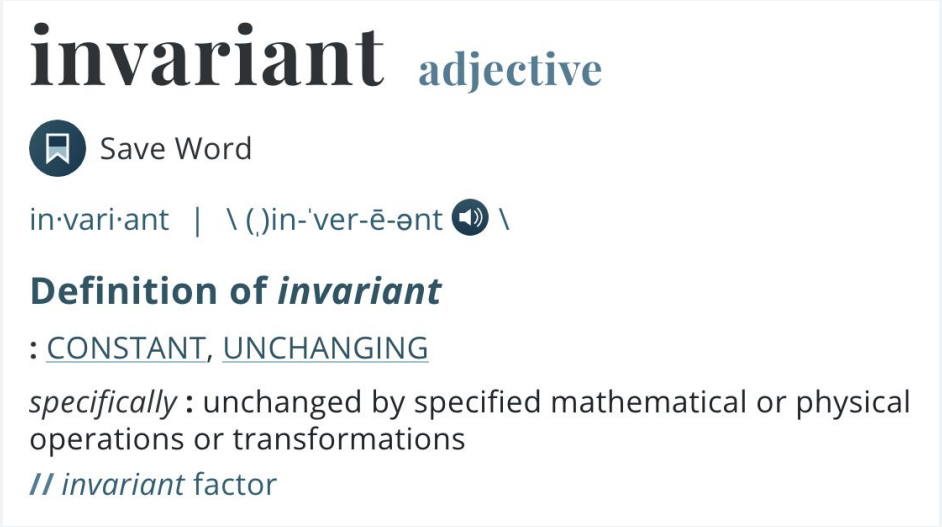
- **Stress the program with random inputs***
  - Most basic fuzzer: randomly type on your keyboard
- **Fuzzing is well established in traditional software security**
  - AFL, Libfuzzer, go-fuzz, ..

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

Save Word

in·vari·ant | \ (ˌ)in-ˈver-ē-ənt \

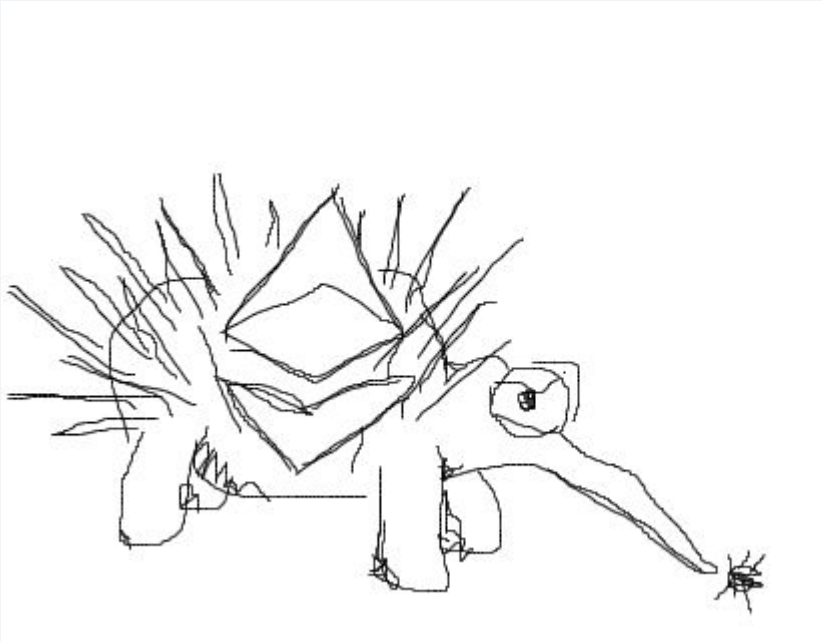
Definition of *invariant*

: CONSTANT, UNCHANGING

specifically : unchanged by specified mathematical or physical operations or transformations

// invariant factor

Echidna



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

Exercises

Step 0: Install Echidna

Mac OS X

```
brew install echidna
```

Linux

```
nix-env -i -f
https://github.com/crytic/echidna/tarball/master
```

Otherwise

Download binaries from [crytic/echidna]

Exercise 1

- program-analysis/echidna/Exercise-1.md
- Exercise-1.md
- Goal: implement basic arithmetic checks
- Note: use Solidity 0.7 (see solc-select if needed)

First: try without the template!

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;
}
```

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;
    }
```

Echidna - Workflow

Smart Contract Code

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

Property Invariant

```
require(balance[msg.sender] <= totalSupply);
```

input

Echidna Tests

Can Echidna break the invariant?

Echidna - Workflow

Smart Contract Code

```
pragma solidity 0.7.0;
contract Token {   //address(0x0)       1
   function transfer(address to, uint value)
public{
        balances[msg.sender] -= value;
        balances[to] += value;
    }
}
```

Property Invariant

```
require(balance[msg.sender] <= totalSupply);
```

input

Echidna Tests

```
transfer(0x0, 1)
```

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`
    - Sender balance should decrease by amount
    - Receiver balance should increase by 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

Identify invariants: tokens

- **ERC20.total_supply**
  - No user should have a balance > total_supply
- **ERC20.transfer:**
  - After calling `transfer`
    - Sender balance should decrease by amount
    - Receiver balance should increase by amount
  - If self transfer is attempted => identical balance
  - If insufficient funds => tx 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 target**
- **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 or inheritance
  - Complex initialization: leverage your unit test/deployment scripts [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);
    }
}
```

Exercise 2

Exercise 2

- program-analysis/echidna/Exercise-2.md
- Exercise-2.md
- Goal: BREAK MORE STUFF!
- Note: use Solidity 0.8.0 (see solc-select if needed)

We'll work together first ;)

Where to focus?

Where to focus?

- **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?

Where to focus?

- **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

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)`
  - Is transfer additive-like?
    - `transfer(to, v0), transfer(to, v1) ?= transfer(to, v0 + v1)?`

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**

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*
  - Watch out for development on Medusa