TRAIL OFBITS

Before starting

- git clone https://github.com/crytic/building-secure-contracts
- git checkout defi101



Building secure contracts: How to fuzz like a pro

Who are we?

- Josselin Feist (@montyly)
- Nat Chin (@0xicingdeath)
- Justin Jacob (<u>@technovision99</u>)



ToB Twitter list

- 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 to find bugs?
- What is property based testing?
- Exercises: simple and more advanced fuzzing
- How to define good invariants?
- Comparison with similar tools

```
/// @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

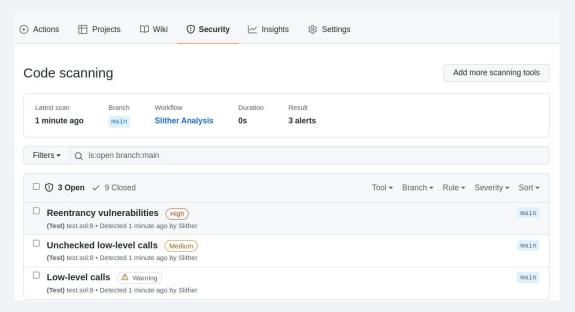
```
function test_buy(uint256 tokens_to_receive, uint256 ether_to_send) public {
   uint256 pre_buy_balance = token.balanceOf(address(this));
   mock.buy.call{value: ether_to_send)(tokens_to_receive);
   assert(token.balanceOf(address(this)) == pre_buy_balance + tokens_to_receive)
}
```

Manual review

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

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

Slither Action



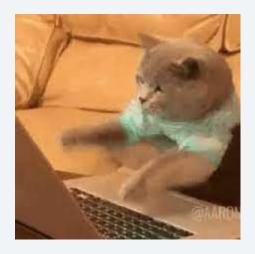
https://github.com/crytic/slither-action

- Semi automated analysis
 - Benefits
 - Great for logic-related bugs
 - Limitations
 - Require human in the loop
 - Example: Property based testing with <u>Echidna</u>

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 fuzzers generally detect crashes
 - Smart contracts don't (really) have crashes
- Property based testing
 - User defines invariants
 - Fuzzer generates random inputs
 - Check whether specified "incorrect" state can be reached
- "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

II invariant factor

Invariant - Token's total supply

User balance never exceeds total supply

Echidna

- Smart contract fuzzer
- Open source:
 github.com/crytic/echidna
- Heavily used in audits & mature codebases
- Focused in easy to use
 - Solidity invariants
 - Github action
 - All compilation frameworks

Public use of Echidna

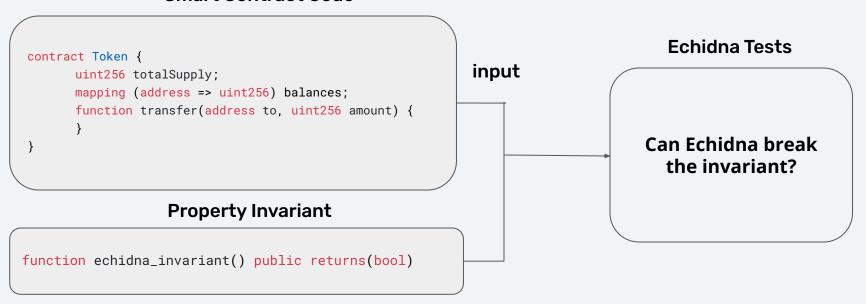
Property testing suites

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 - Overview

Smart Contract Code



Exercises

Exercise 1

- git clone https://github.com/crytic/building-secure-contracts
- git checkout defi101
- Open program-analysis/echidna/exercises/Exercise-1.md

Goal: check if total supply invariant holds

Notes:

- Use Solidity 0.8 (see solc-select if needed)
- Try without the template!



Exercise 1 - Target

```
contract Token is Ownable, Pausable {
    mapping(address => uint256) public balances;

    function transfer(address to, uint256 value) public whenNotPaused {
        // unchecked to save gas
        unchecked {
            balances[msg.sender] -= value;
            balances[to] += value;
        }
    }
}
```

Exercise 1 - Template

```
contract TestToken is Token {
   address echidna_caller = msg.sender;
   constructor() public {
       balances[echidna_caller] = 10000;
   // add the property
```

```
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;</pre>
```

\$ echidna solution.sol

```
echidna_test_balance: FAILED! with ReturnFalse

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

```
contract Token is Ownable, Pausable {
    mapping(address => uint256) public balances;

    function transfer(address to, uint256 value) public whenNotPaused {
        // unchecked to save gas
        unchecked {
            balances[msg.sender] -= value;
            balances[to] += value;
        }
    }
}
```

Exercise 2

- git clone https://github.com/crytic/building-secure-contracts
- git checkout defi101
- Open <u>program-analysis/echidna/exercises/Exercise-2.md</u>

Goal: can you unpause the system?

Note: try without the template!



Exercise 2 - Target

```
contract Ownable {
   address public owner = msg.sender;

function Owner() public {
    owner = msg.sender;
}

modifier onlyOwner() {
   require(owner == msg.sender);
   -;
}
```

```
contract Pausable is Ownable {
    bool private _paused;
   function paused() public view returns
(bool) {
        return _paused;
    function pause() public onlyOwner {
        _paused = true;
    function resume() public onlyOwner {
        _paused = false;
```

```
contract TestToken is Token {
   constructor() {
       paused();
       owner = 0x0; // lose ownership
   // add the property
```

```
contract TestToken is Token {
  constructor() {
       paused();
       owner = 0x0; // lose ownership
  function echidna_no_transfer() view returns(bool) {
       return is paused == true;
```

\$ echidna-test solution.sol

```
echidna_no_transfer: FAILED! with ReturnFalse

Call sequence:
1.0wner()
2.resume()
```

```
contract Ownership{
  address owner = msg.sender;
  function Owner(){
    owner = msg.sender;
  }
  modifier isOwner(){
    require(owner == msg.sender);
    _;
  }
}
```

```
contract Pausable is Ownership{
   bool is_paused;
  modifier ifNotPaused(){
            require(!is_paused);
    function paused() isOwner public{
       is_paused = true;
    function resume() isOwner public{
       is_paused = false;
```

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

- Start early, before starting to code
- 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

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

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 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

```
/// @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)
```

```
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);
}
```

Echidna APIs

Echidna APIs

- Boolean properties
- Assertion
- Dapp/foundry API

<u>secure-contracts.com/program-analysis/echidna/basic/testing-m</u> <u>odes.html</u>

Boolean properties

- Most of our examples so far default mode
- echidna_something() returns(bool)
- Benefits
 - Easy to use
 - Invariants easy to find
 - No side effects are kept
- Limitations
 - No parameters
 - Revert is a failure
 - No coverage on echidna_something

Assertion

- Solidity assert()
- Benefits
 - Simpler for function introspection
 - Code coverage
- Limitations
 - Difficult to use if the codebase misuse assert
 - Must be careful where the assert are added to not break the original code

Dapp/foundry

- setUp() + checking for reverting function
- Benefits
 - Compatible with foundry
- Limitations
 - Require to handle reverts (e.g. using FOUNDRY::ASSUME)

Exercise 4(*) - Assertion

- git clone https://github.com/crytic/building-secure-contracts
- git checkout defi101
- Open <u>defi101/program-analysis/echidna/exercises/Exercise-4.md</u>

Goal: check if total supply invariant holds with assertion

First: try without the template!

(*) - no exercise 3 today

```
contract TestToken is Token {
    function transfer(address to, uint256 value) public override {
        uint256 oldBalanceFrom = balances[msg.sender];
        uint256 oldBalanceTo = balances[to];

        super.transfer(to, value);

        assert(balances[msg.sender] <= oldBalanceFrom);
        assert(balances[to] >= oldBalanceTo);
    }
}
```

Exercise 4 - Assertion

The assertions can be kept in the code

```
assert(balances[msg.sender] <= oldBalanceFrom);
assert(balances[to] >= oldBalanceTo);
```

- Benefits
 - Onchain safeguard
 - Explicit invarians/post condition
- Drawbacks
 - Gas
 - Can be complex and have bugs

Composability

All contract

- By default, Echidna focuses on one contract
- Enable the all-contracts allows Echidna to work on composability issue:
 - Use command-line flag --all-contracts
 - Or use allContracts: true in the config file

Exercise 5 - Damn-Vulnerable-Defi

- git clone https://github.com/crytic/building-secure-contracts
- git checkout defi101
- Open program-analysis/echidna/exercises/Exercise-5.md

Goal: let echidna solves the NaiveReceiver challenge

First: try without the hints

Exercise 5 - Description

- Two contracts
 - NaiveReceiverLenderPool: allow to take a flash loan for a fee
 - FlashLoanReceiver: user's contract taking flash loan
- The user deploys a FlashLoanReceiver with 10 eth. Can you drain the funds?

Exercise 5 - Target (NaiveReceiverLenderPool)

```
function flashLoan(address borrower, uint256 borrowAmount) external nonReentrant {
   uint256 balanceBefore = address(this).balance;
    require(balanceBefore >= borrowAmount, "Not enough ETH in pool");
    require(borrower.isContract(), "Borrower must be a deployed contract");
    // Transfer ETH and handle control to receiver
    borrower.functionCallWithValue(
        abi.encodeWithSignature(
            "receiveEther(uint256)",
            FIXED FEE
        borrowAmount
    );
    require(
        address(this).balance >= balanceBefore + FIXED FEE,
        "Flash loan hasn't been paid back"
    );
```

Exercise 5 - Target (FlashLoanReceiver)

```
// Function called by the pool during flash loan
function receiveEther(uint256 fee) public payable {
    require(msg.sender == pool, "Sender must be pool");
    uint256 amountToBeRepaid = msg.value + fee;
    require(address(this).balance >= amountToBeRepaid, "Cannot borrow that much");
    executeActionDuringFlashLoan();
    // Return funds to pool
    pool.sendValue(amountToBeRepaid);
```

Exercise 5 - Initialization

```
before(async function () {
    /** SETUP SCENARIO - NO NEED TO CHANGE ANYTHING HERE */
    [deployer, user, attacker] = await ethers.getSigners();
    const LenderPoolFactory = await ethers.getContractFactory('NaiveReceiverLenderPool', deployer);
    const FlashLoanReceiverFactory = await ethers.getContractFactory('FlashLoanReceiver', deployer);
    this.pool = await LenderPoolFactory.deploy();
    await deployer.sendTransaction({ to: this.pool.address, value: ETHER_IN_POOL });
    expect(await ethers.provider.getBalance(this.pool.address)).to.be.equal(ETHER_IN_POOL);
    expect(await this.pool.fixedFee()).to.be.equal(ethers.utils.parseEther('1'));
    this.receiver = await FlashLoanReceiverFactory.deploy(this.pool.address);
    await deployer.sendTransaction({ to: this.receiver.address, value: ETHER_IN_RECEIVER });
    expect(await ethers.provider.getBalance(this.receiver.address)).to.be.equal(ETHER IN RECEIVER);
});
```

Config file

```
# 10,000 ether is placed in the NaiveReceiverEchidna
contract.
balanceContract: 100000000000000000000
# Allow for multi-abi use
allContracts: true
```

```
// We will send ETHER_IN_POOL to the flash loan pool.
    uint256 constant ETHER_IN_POOL = 1000e18;
    // We will send ETHER_IN_RECEIVER to the flash loan receiver.
    uint256 constant ETHER_IN_RECEIVER = 10e18;
    // Setup echidna test by deploying the flash loan pool and receiver and sending them
some ether.
    constructor() payable {
        pool = new NaiveReceiverLenderPool();
        receiver = new FlashLoanReceiver(payable(address(pool)));
        payable(address(pool)).sendValue(ETHER_IN_POOL);
        payable(address(receiver)).sendValue(ETHER_IN_RECEIVER);
    // We want to test whether the balance of the receiver contract can be decreased.
    function echidna_test_contract_balance() public view returns (bool) {
        return address(receiver).balance >= 10 ether;
```

```
echidna_test_contract_balance: FAILED! with ReturnFalse

Call sequence:
1.flashLoan(0x62d69f6867a0a084c6d313943dc22023bc263691,1000000000000000001)
```

Access controls issue

- **Anyone** can trigger the flash loan on the user contract
- An attacker can do flash loans on behalf of the receiver's owner and drain the funds through the fees

Exercise 6 - Damn-Vulnerable-Defi

- git clone https://github.com/crytic/building-secure-contracts
- git checkout defi101
- Open program-analysis/echidna/exercises/Exercise-6.md

Goal: let echidna solves the Unstoppable challenge

First: try without the hints

Exercise 6 - Description

- Two contracts
 - UnstoppableLender: allow to take a flash loan and do a callback on the caller
 - ReceiverUnstoppable: user callback example
- Can you prevent UnstoppableLender from working?

Exercise 6 - Target (UnstoppableLender)

```
function flashLoan(uint256 borrowAmount) external nonReentrant {
   require(borrowAmount > 0, "Must borrow at least one token");
    uint256 balanceBefore = damnValuableToken.balanceOf(address(this)):
    require(balanceBefore >= borrowAmount, "Not enough tokens in pool");
    // Ensured by the protocol via the `depositTokens` function
    assert(poolBalance == balanceBefore);
   damnValuableToken.transfer(msg.sender, borrowAmount);
    IReceiver(msg.sender).receiveTokens(address(damnValuableToken), borrowAmount);
    uint256 balanceAfter = damnValuableToken.balanceOf(address(this));
    require(balanceAfter >= balanceBefore, "Flash loan hasn't been paid back");
```

Exercise 6 - Initialization

```
before(async function () {
    /** SETUP SCENARIO - NO NEED TO CHANGE ANYTHING HERE */

    [deployer, attacker, someUser] = await ethers.getSigners();

const DamnValuableTokenFactory = await ethers.getContractFactory('DamnValuableToken', deployer);

const UnstoppableLenderFactory = await ethers.getContractFactory('UnstoppableLender', deployer);

this.token = await DamnValuableTokenFactory.deploy();
this.pool = await UnstoppableLenderFactory.deploy(this.token.address);

await this.token.approve(this.pool.address, TOKENS_IN_POOL);
await this.pool.depositTokens(TOKENS_IN_POOL);

await this.token.transfer(attacker.address, INITIAL_ATTACKER_TOKEN_BALANCE);
```

Exercise 6 - Initialization

```
expect(
    await this.token.balanceOf(this.pool.address)
).to.equal(TOKENS_IN_POOL);

expect(
    await this.token.balanceOf(attacker.address)
).to.equal(INITIAL_ATTACKER_TOKEN_BALANCE);

// Show it's possible for someUser to take out a flash loan
    const ReceiverContractFactory = await ethers.getContractFactory('ReceiverUnstoppable', someUser);
    this.receiverContract = await ReceiverContractFactory.deploy(this.pool.address);
    await this.receiverContract.executeFlashLoan(10);
});
```

Config file

```
# The deployer and sender must be the same for this example.
# The deployer is the 'attacker' and is sent INITIAL_ATTACKER_BALANCE
# The actual value does not matter, as long as they are the same
deployer: '0x30000'
# Sender must be the same so that it can use the attacker balance to try to break
the invariant.
sender: ['0x30000']
# Allow for multi-abi use
allContracts: true
```

```
// We will send ETHER_IN_POOL to the flash loan pool.
uint256 constant ETHER_IN_POOL = 1000000e18;
// We will send INITIAL_ATTACKER_BALANCE to the attacker (which is the deployer)
of this contract.
uint256 constant INITIAL_ATTACKER_BALANCE = 100e18;
DamnValuableToken token;
UnstoppableLender pool;
// Setup echidna test by deploying the flash loan pool, approving it for token
transfers, sending it tokens, and sending the attacker some tokens.
constructor() public payable {
    token = new DamnValuableToken();
    pool = new UnstoppableLender(address(token));
    token.approve(address(pool), ETHER_IN_POOL);
    pool.depositTokens(ETHER_IN_POOL);
    token.transfer(msg.sender, INITIAL_ATTACKER_BALANCE);
```

```
// This is the callback function for flash loan receivers.
function receiveTokens(address tokenAddress, uint256 amount) external {
    require(msg.sender == address(pool), "Sender must be pool");
    // Return all tokens to the pool
    require(
        IERC20(tokenAddress).transfer(msg.sender, amount),
        "Transfer of tokens failed"
    );
// This is the Echidna property entrypoint.
// We want to test whether flash loans can always be made.
function echidna_testFlashLoan() public returns (bool) {
    pool.flashLoan(10);
    return true;
```

 The pool require an exact balance equality - sending token to directly to the pool will break this requirements

```
// Ensured by the protocol via the `depositTokens` function
assert(poolBalance == balanceBefore);
```

Comparison with similar tools

Other fuzzers

- Inbuilt in dapp, brownie, foundry, ...
- Might be easier for simple test, however
 - Less powerful
 - 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

Medusa

- https://github.com/crytic/medusa
- Rewrite of Echidna in Go
- Still experimental, but we are looking for feedback

Conclusion

Conclusion

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

Additional slides

- 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)?

- 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?

- 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