Please perform the following tasks. If you find any ambiguities, clearly state any assumptions you make and solve the task under those assumptions.

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.

Verification

Look at the code given in Game.sol. Does `attempt` adhere to the following formal function specification:

- Precondition: `started(Game.attempt(guess), started && msg.value > cost)`
- Postcondition: `finished(Game.attempt(guess), value = fsum(Game.attempt(guess), guess))`

If not, explain why.

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.

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.

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;
    }
}
```