Name: Penugonda pranav

Registration number: 12104850

Roll number: 18

Section: k21cs

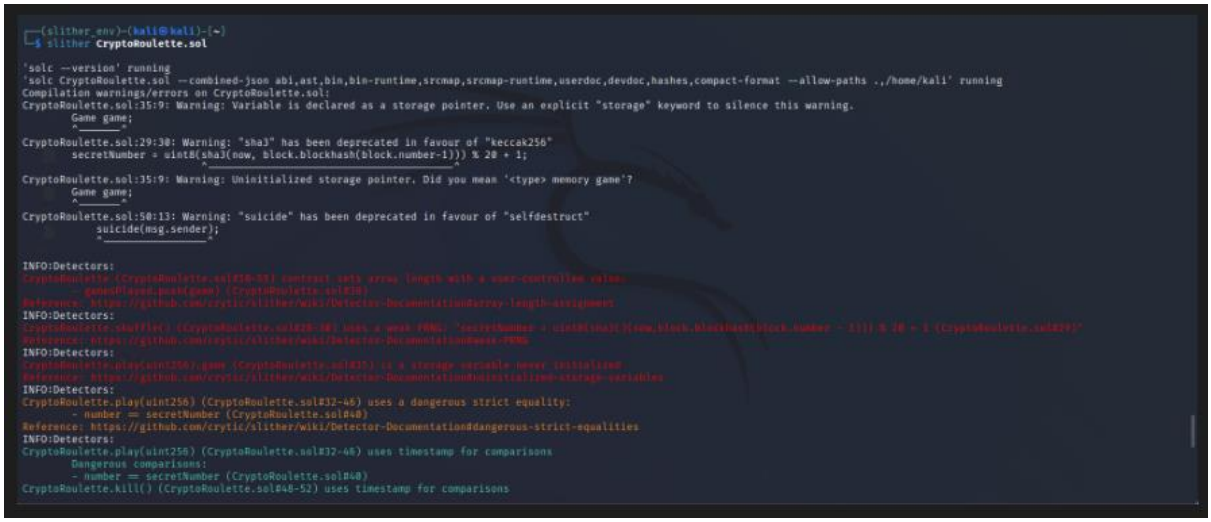
Subject: csc403

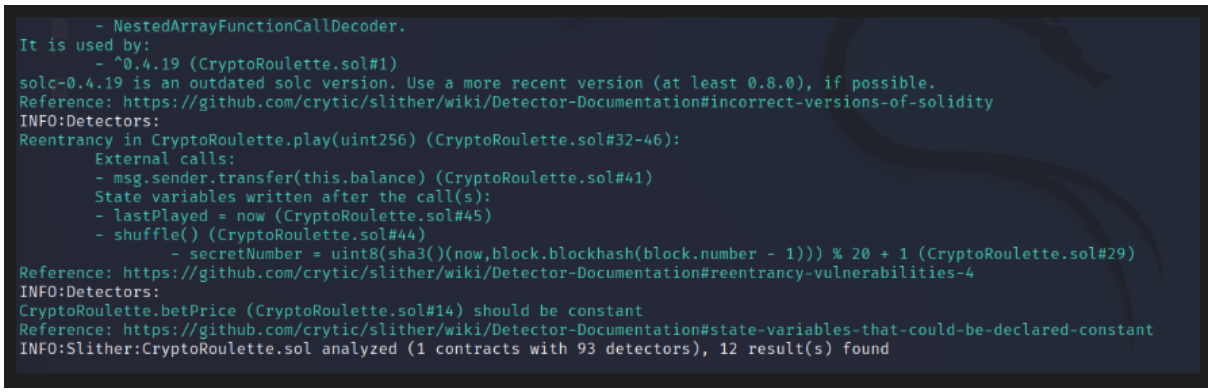
Using an automated security audit tool Slither, perform an audit of following smart contract. Identify and document security vulnerabilities found by the tool, describe how each could be exploited, and propose fixes for each vulnerability in an audit report format.

```
pragma solidity ^0.4.19;


// CryptoRoulette
//
// Guess the number secretly stored in the blockchain and win the whole contract balance!
// A new number is randomly chosen after each try.
//
// To play, call the play() method with the guessed number (1-20).  Bet price: 0.1 ether


contract CryptoRoulette {


  uint256 private secretNumber;
  uint256 public lastPlayed;
  uint256 public betPrice = 0.1 ether;
  address public ownerAddr;


  struct Game {
    address player;
    uint256 number;
  }
  Game[] public gamesPlayed;


  function CryptoRoulette() public {
    ownerAddr = msg.sender;
    shuffle();
  }


  function shuffle() internal {
    secretNumber = uint8(sha3(now, block.blockhash(block.number-1))) % 20 + 1;
```

```
    }

    function play(uint256 number) payable public {
        require(msg.value >= betPrice && number <= 10);

        Game game;
        game.player = msg.sender;
        game.number = number;
        gamesPlayed.push(game);

        if (number == secretNumber) {
            msg.sender.transfer(this.balance);
        }

        shuffle();
        lastPlayed = now;
    }

    function kill() public {
        if (msg.sender == ownerAddr && now > lastPlayed + 1 days) {
            suicide(msg.sender);
        }
    }

    function() public payable { }
}
```

Slither output





Audit Report: CryptoRoulette Smart Contract

1. Contract Overview

The CryptoRoulette contract allows users to guess a secret number stored on-chain for a chance to win the contract's balance. It includes functionality for playing the game, managing funds, and self-destruction. The audit identified multiple vulnerabilities, including reentrancy risks, reliance on outdated and insecure Solidity features, and poor randomness generation.

Audit Report for CryptoRoulette Contract

---

1. Weak PRNG for Secret Number Generation

- **Description:**
  The contract uses uint8(sha3(now, block.blockhash(block.number-1))) % 20 + 1 for generating the secretNumber. This method is vulnerable because:
  - block.timestamp (now) can be influenced slightly by miners.
  - block.blockhash is predictable for the current or recent blocks.
  - sha3 is deprecated in favor of keccak256.
- **Exploit Scenario:**
  Attackers could predict the secretNumber by monitoring block data and timestamp, enabling them to consistently win the game.
- **Fix:**
  Use keccak256 with a better entropy source that includes user-specific and random elements. For example:

secretNumber = uint256(keccak256(abi.encodePacked(block.timestamp, blockhash(block.number - 1), msg.sender))) % 20 + 1;

---

2. Uninitialized Storage Pointer

- **Description:**
  In the play() function, the line Game game; declares a Game storage pointer without initialization. This points to slot 0 in storage, potentially overwriting critical data**.**
- **Exploit Scenario:**
  A malicious player could exploit this bug to overwrite existing storage variables, corrupting the contract's state.
- **Fix:**
  Avoid declaring uninitialized storage variables. Directly push to the gamesPlayed array**:**

gamesPlayed.push(Game({player: msg.sender, number: number}));

---

3. Deprecated Functions (sha3, suicide)

- **Description:**
  The contract uses sha3 and suicide, both deprecated functions:
    - sha3 should be replaced by keccak256.
    - suicide is replaced by selfdestruct. However, selfdestruct itself has been deprecated starting from the Cancun hard fork.
- **Exploit Scenario:**
  While these functions work in older Solidity versions, their behavior is outdated and could lead to unexpected results in newer EVM versions.
- **Fix:**
    - Replace sha3 with keccak256.
    - Instead of selfdestruct, implement a "disable" mechanism to stop interactions and transfer remaining funds to the owner.

---

4. Reentrancy Vulnerability

- **Description:**
  The contract calls msg.sender.transfer(this.balance) inside the play() function. Since Ether transfer happens before state updates, this opens the door for reentrancy attacks.
- **Exploit Scenario:**
  An attacker could call the play() function through a malicious contract that re-enters the play() function repeatedly, draining the contract balance.
- **Fix:**
  Update state variables before making external calls. Refactor the play() function to:

```
bool isWinner = (number == secretNumber);

lastPlayed = block.timestamp;

shuffle();

if (isWinner) {

   payable(msg.sender).transfer(address(this).balance);

}
```

---

5. Dangerous Equality Check (number == secretNumber)

- **Description:**
  The number == secretNumber comparison directly determines whether the player wins. If combined with weak PRNG, this is easily exploitable.

- **Exploit Scenario:**
  Attackers could exploit predictable PRNG and the strict equality condition to ensure they always win.
- **Fix:**
  Strengthen PRNG as described earlier. Consider adding additional randomness (e.g., from a trusted oracle).

---

6. Timestamp Dependence

- **Description:**
  The contract uses now (or block.timestamp) in two places:
  - In PRNG (sha3(now, ...))
  - In the kill() function's condition (now > lastPlayed + 1 days).

Miners can manipulate the timestamp within a certain range, introducing vulnerabilities.

- **Exploit Scenario:**
  - Miners could influence the PRNG outcome by choosing favorable timestamps.
  - They could also manipulate lastPlayed to prevent or prematurely allow contract termination.
- **Fix:**
  - Replace now with block.timestamp.
  - Use block-based conditions instead of time-based ones (e.g., block.number).

---

7. Lack of Contract Activity Lock

- **Description:**
  The contract has no mechanism to prevent further interaction once the owner intends to terminate it. This is exacerbated by the deprecation of selfdestruct.
- **Exploit Scenario:**
  Attackers could continue interacting with the contract, draining Ether, even after the owner attempts to shut it down.
- **Fix:**
  Introduce an isActive state variable to enable or disable the contract:

```
bool public isActive = true;


modifier contractIsActive() {
  require(isActive, "Contract is disabled");
  _;
```

}

---

8. Inefficient Constants

- **Description:**
  The variable betPrice is declared as a regular state variable but never changes. This unnecessarily consumes storage gas.
- **Fix:**
  Declare it as a constant:

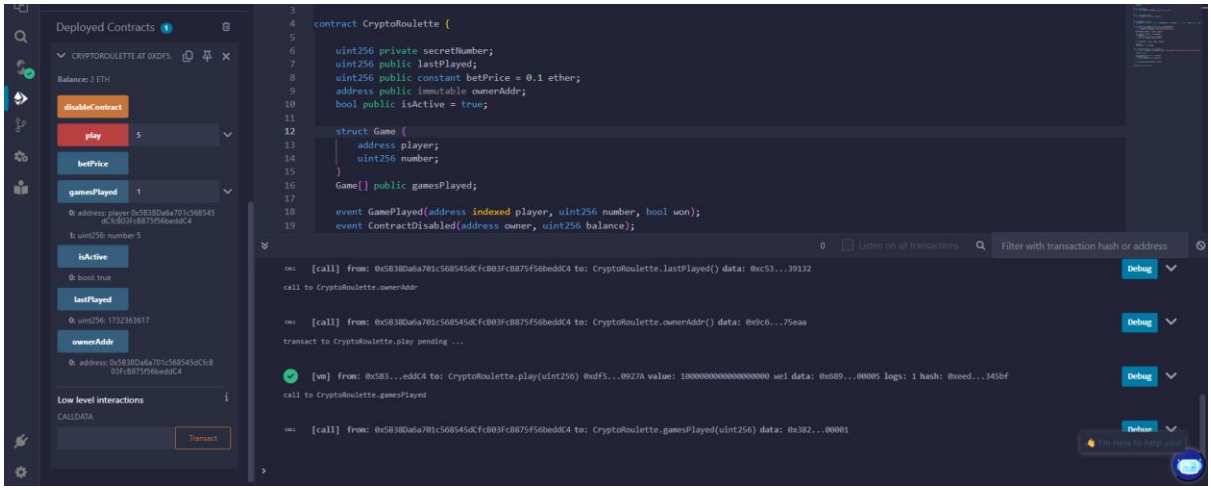
uint256 public constant betPrice = 0.1 ether;

---

9. Solidity Version (^0.4.19)

- **Description:**
  The contract uses Solidity ^0.4.19, which contains known vulnerabilities (e.g., reentrancy and uninitialized storage variables).
- **Fix:**
  Use a more recent Solidity version (e.g., ^0.8.0) to leverage modern security checks, including overflow protection and better error handling.

Output:



Check account to confirm bet amount is credited or not